Der ns-2 Netzwerksimulator

(Eine Dokumentation zum Einsatz von ns-2 im Unterricht auf Basis der über http://www.isi.edu/nsnam/ns/ erreichbaren Unterlagen)

Version vom 13. 1. 2004

Aktuelle Version und online Material: http://www.welzl.at/tools/ns

Autoren: **Gernot Gebhart, Christian Kienzl, Stefan Kirchmair**

Betreuung / Überarbeitung: **Michael Welzl**

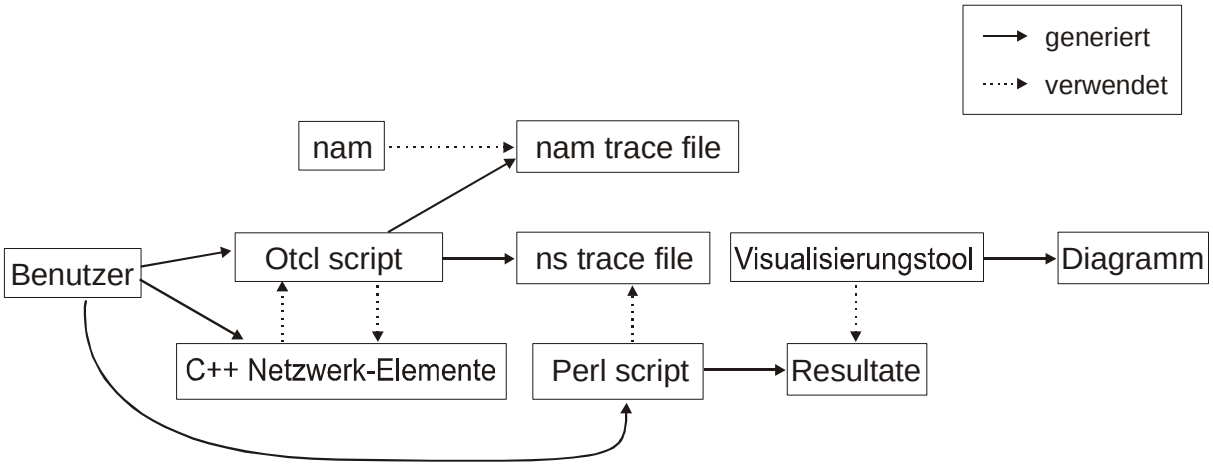
„Beta-Tester“: **Übungsleiter und Tutoren „Computernetzwerke“ im WS 03/04**

Inhaltsverzeichnis

Kapitel 1 : Einführung

Der **ns** (eigentlich „**ns**-2“) Netzwerksimulator ist ein state-of-the-art Programm zur Simulation von paketvermittelten Computernetzen. Er hält sich zwar nicht strikt an Formalismen, kann aber als ein Diskreter Event Simulator (DEVS) betrachtet werden: das Senden eines Pakets zu einer bestimmten Zeiteinheit ist ebenso ein „Event“ wie das Empfangen eines solchen; Events werden in einer Script-Datei spezifiziert, am Ende der Simulation liegt eine Ergebnisdatei („trace file“) vor, aus der sich Resultate ableiten lassen. Scripts werden in der Sprache „**OTcl**“ (eine objektorientierte Erweiterung von **Tcl**) geschrieben; die Interna (Protokolle, ..) sind jedoch in C++ realisiert. Die Idee hinter dem Einsatz dieser beiden Sprachen war die Entwicklung von Netzwerkverfahren (C++) von der eigentlichen Simulationstätigkeit (**OTcl**) zu trennen – es soll möglich sein, schnell Simulationen auszuprobieren, ohne den gesamten Netzwerksimulator neu übersetzen zu müssen.

Folgende Abbildung zeigt einen Überblick über die Arbeitsweise mit **ns**:



Die Verwendung von **ns**

ns wurde von Forschern für Forscher geschaffen; demnach vermischt sich in seiner Dokumentation der interne Aufbau (C++ Elemente) mit der Benutzung von Netzwerkelementen (**OTcl**). Dieses Dokument wurde geschaffen, um den Einsatz von **ns** im Unterricht zu erleichtern: **ns** soll nur benutzt und nicht verändert werden, d.h. es werden keine Interna (C++) erläutert, sondern es geht ausschließlich um die Verwendung der verfügbaren Mechanismen.

ns ist sehr groß und wächst ständig. Darum ist es unmöglich, alle relevanten Details in diesem Dokument abzudecken. Für jegliche weitere Dokumentation sei daher auf die Startseite des open-source-Projekts verwiesen: http://www.isi.edu/nsnam/ns

Benutzen dieser Dokumentation

In den folgenden Kapiteln wird in die Scriptsprache **Tcl** und in die nötigen Konzepte des Netzwerksimulators (Knoten, Agenten, Links etc.) eingeführt. Da **ns** sehr umfangreich ist, ist diese Dokumentation entsprechend lang.

Es hat sich in der Praxis gezeigt, daß oft auch eine schnelle, Tutorial-artige Einführung genügt, um erste Schritte mit dem **ns** Netzwerksimulator zu machen – selbst, wenn man weder **Tcl** noch **OTcl** kennt und noch nie mit einem Simulator gearbeitet hat. Darum wurde für ungeduldige Leser auf Basis des bekannten „Marc Greis Tutorials“ eine Art Schnellstart-Einführung erstellt; diese findet sich im Anhang. Zur Realisierung etwas aufwändigerer Simulationen führt jedoch vermutlich kein Weg am Studium der folgenden Kapitel vorbei; dabei kann Anhang A als Referenzkapitel für Beispiele dienen.

Installation

Der Netzwerksimulator kann von folgender Adresse herunter geladen werden:
http://www.isi.edu/nsnam/ns/ns-build.html

Unter Linux empfiehlt sich die Installation des allinone-Paketes; die aktuelle Version zum Zeitpunkt, an dem diese Dokumentation geschrieben wurde, ist:
http://www.isi.edu/nsnam/dist/ns-allinone-2.26.tar.gz
Dieses Paket beinhaltet u.a. die Programme **ns**, **nam** und **xgraph** sowie den **Tcl** Interpreter.

ns wurde grundsätzlich unter und für Unix entwickelt; es gibt eine Gruppe von Forschern, die regelmäßig die aktuelle Version auf Windows portiert; nicht immer arbeiten jedoch gerade die letzten Versionen einwandfrei. Unter Windows funktioniert die Version
http://www.isi.edu/nsnam/dist/binary/ns-2.1b5.exe von **ns** und
http://www.isi.edu/nsnam/dist/binary/nam-1.0a11a-win32.exe von **nam**

Sollten bei der Installation Probleme auftreten steht eine Homepage für die am häufigsten auftretenden Probleme zur Verfügung.
http://www.isi.edu/nsnam/ns/ns-problems.html
http://www.isi.edu/nsnam/ns/ns-lists.html

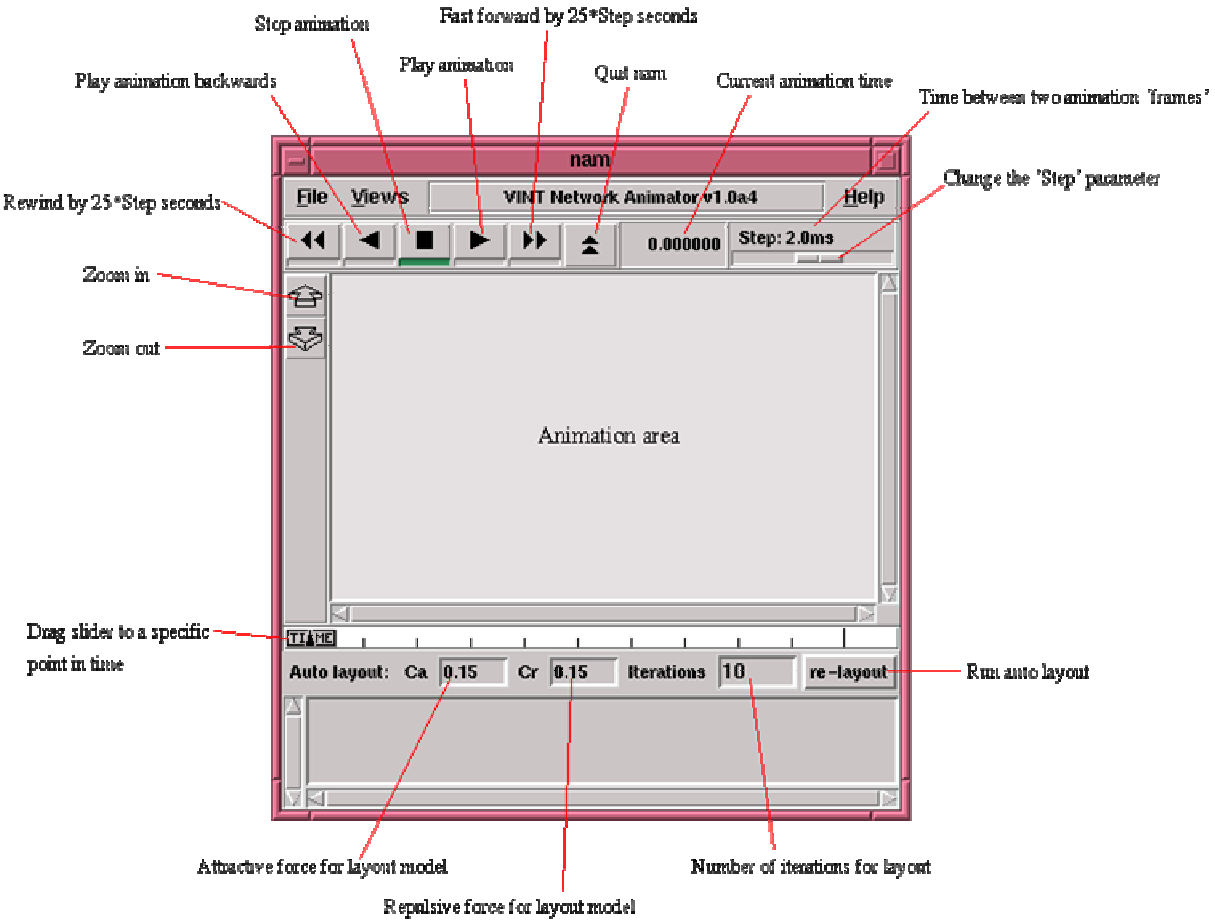
Benötigte Programme

ns

Der eigentliche Netzwerksimulator. Mit diesem Programm wird ein in einem **tcl**-Skript spezifiziertes Netzwerk simuliert. Man kann **ns** mit dem Befehl **ns** <tclscript> ausführen. Eine weitere (eher selten genutzte) Möglichkeit ist es, **ns** ohne irgendwelche Parameter zu starten, und dann in der shell weitere Befehle auszuführen.

nam

Dient zur graphischen Darstellung der Simulation *nach* deren Ausführung. **nam** kann mit dem Befehl **nam** <namfile> gestartet werden, wobei <namfile> ein **nam** Trace-File ist, das von **ns** generiert werden muss.



Screenshot von **nam**

xGraph

Dient zur graphischen Darstellung von Simulationsergebnissen.

Kapitel 2: Einführung in Tcl/Tk

Was ist Tcl und was ist Tk ?

Tcl steht für „*Tool Command Language*“ und ist eine Programmiersprache, die von einem Interpreter abgearbeitet (zur Laufzeit interpretiert) wird. Sprachen, für die das auch gilt sind z.B. die UNIX-Shells, Perl etc. Die Idee zu **Tcl** entstand in den frühen 80er Jahren, als Prof. John K. Ousterhout an der University of California at Berkeley mit seinen Studenten an Tools zum Design von integrierten Schaltkreisen arbeitete. Durch das Fehlen einer universellen Allzwecksprache musste für jedes Tool eine eigene Befehlssprache entwickelt werden, was zu einem nicht unerheblichen Aufwand führte.

Daher kam die Idee auf, eine solche Sprache als Library von C-Funktionen zu schreiben, die wieder verwendbar und beliebig erweiterbar wäre. Im Frühjahr 1988 begann Ousterhout mit der Arbeit, deren Ergebnisse zunächst **Tcl** und später **Tk** (Toolkit - ein auf **Tcl** basierendes Toolkit für X11) waren und die bis heute vorangehen.

Tcl ist als Interpretersprache konzipiert, die eine Shell zur Ausführung der **Tcl**-Skripte benötigt, wie `Tclsh` (**Tcl** - Shell) für **Tcl** und **`wish`** (Windowing shell) für **Tcl/Tk**. Sie ermöglicht es mittels der übliche Programmierkonstrukte, wie z.B. Variablen, Schleifen und Prozeduren, neue Applikationen zu schreiben und bestehende zu erweitern. Tatsächlich besteht der Interpreter aus einer einfachen C-Library, die den Programmen hinzugelinkt werden.

Ein wichtiges Ziel von **Tcl** ist es nunmehr komplexe Anwendungen auf einem hohen Abstraktionsniveau entwickeln zu können, und im Vergleich zu C-Programmen unter Umständen ein Vielfaches an Codelänge einzusparen und gegebenenfalls weniger Wissen um die interne Funktionsweise des zugrunde liegenden Systems (bspw. X11) zu benötigen.

Ferner kann man aufgrund der Tatsache, dass **Tcl** eine Interpretersprache ist, schnell, ohne großen Compilier-Aufwand neue Anwendungen ausprobieren ("Rapid-Prototyping"). Es sei direkt angefügt, dass dies sicher auch der größte Nachteil von **Tcl** ist, da selbstverständlich jegliche in einer Interpretersprache geschriebene Applikation deutlich langsamer ist, als das Pendant in einer kompilierten Version.

Neben diesen zentralen, zum größten Teil positiven Eigenschaften erhält der Entwickler einige weitere nützliche Eigenschaften, wenn er mit **Tcl** entwickelt. Die Sprache ist im Kern recht knapp gehalten, aber trotzdem mächtig, leistungsstark und bleibt ständig benutzerfreundlich.

Eine weitere große Stärke von **Tcl** ist die leichte Erweiterbarkeit. So lassen sich neue Befehle ganz einfach in C schreiben, und können dann in **Tcl** eingebunden werden. So sind die bereits erwähnten meist schon in der Distribution enthaltenen. Pakete von **Tk** und **Extended Tcl** selbst enthalten auch solche C-Libraries.

Neben all diesen Stärken und Eigenschaften von **Tcl** auf der technischen Seite, ist sicher als überzeugendes Argument für **Tcl** auch dessen Verbreitung (die ja kostenlos vonstatten geht)

und die Entwicklungsfreude der **Tcl**-Benutzgemeinschaft zu nennen, denn gerade dadurch stehen regelmäßig viele neue Tools und Libraries in **Tcl** zur Verfügung.

Distributionen und Literatur

Hier einige der wichtigsten Buch-/Internetliteraturangaben, die in der Erstellung der Kapitel über **Tcl/Tk** Verwendung gefunden haben:

Eine Liste einiger Distributionsadressen findet man unter: [*Tcl*-Distribution[1]]

Eine Liste von Adressen mit *Tcl*-Manual-Seiten findet man unter: [*Tcl*-Man[2]]

Ein **Tcl**-FAQ findet sich unter: [*Tcl*-FAQ[3]]

Einführungen in **Tcl** findet man unter: [Ousterhout94[4]], [Welch95[5]], [Welch-Internet[6]], [Martland[7]]

Zusammenfassungen von Internetseiten zum Themengebiet **Tcl/Tk** sind: [Hobs-*Tcl*-Info[7]],[*Tcl*-Info[8]]

Link zu den Informationen der **Tcl/Tk** Workshops: [Workshop[9]]

[1] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Tcl-Distribution
[2] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Tcl-Man
[3] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Tcl-FAQ
[4] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Ousterhout94
[5] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Welch95
[6] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Welch-Internet
[7] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Hobs-Tcl-Info
[8] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Tcl-info
[9] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Workshop

Die Ausführung von Programmen

Um **Tcl** zu benutzen, muss man also zunächst einen Interpreter starten. Dieser heißt „**Tclsh**“ oder in unserem Umfeld „**wish**“. Die „**wish**“ (Windowing Shell) umfaßt **Tcl** und **Tk**.

Hier ein Beispiel:

$ Tclsh
% expr 2 + 2
4
*% expr 2.3 * 3*
6.9
% expr (7 > 4)
1
% exit
$

Tk steht einfach für „**Tool Kit**“ und stellt eine Erweiterung von **Tcl** dar. **Tk** beinhaltet Befehle, mit denen graphische Benutzeroberflächen erstellt werden können.

Beispiel:

$ wish
% button .b -text "Hello world" -command exit
.b
% pack .b
% $

Tcl-Skripte werden aus einer Shell, wie *`Tclsh`* oder *`wish`* heraus ausgeführt. **`Tclsh`** ist hier die Standardshell von **Tcl**, **`wish`** ist die um **Tk** erweiterte Shell. Startet man diese, so kann man direkt **Tcl**-Befehle ausführen. Die Shells sind prinzipiell nicht mehr, als einfache Auswertungsschleifen, welche die direkte Eingabe von **Tcl**-Befehlen ermöglichen. Das Ergebnis der Befehle wird direkt ausgewertet, und in der nächsten Zeile angezeigt. Die jeweilige Shell startet man, indem man einfach ihren Namen am Kommandoprompt eingibt. Statt die **Tcl**-Befehle einzeln einzugeben, besteht auch die Möglichkeit, das Programm mittels eines beliebigen Editors zu verfassen, abzuspeichern, und dann interaktiv zu starten. Dazu verfügt **Tcl** über den Befehl

`source` ***`filename`*****.**

Das Skript mit dem Namen *`filename`* wird durch diese Eingabe interaktiv abgearbeitet, und die Shell bleibt geöffnet.

Eine zweite Möglichkeit, ein **Tcl**-Skript zu starten, ist, den UNIX-Befehl *`exec`* zu verwenden (bzw. das Skript ausführbar zu machen). Dazu muss jedoch der Pfad der Shell in die erste Zeile des Programms gesetzt werden. Je nachdem wo sich die Shells befinden, müssten die Skripte ähnlich wie folgt beginnen:

```
#!/usr/local/bin/Tcl
```

Falls dies nicht funktioniert, kann man mit dem UNIX-Befehl **find** die Shell suchen, und den Pfadnamen entsprechend ändern.

Zum Konzept von Tcl

Ein **Tcl**-Skript ist eine Reihe von Strings, die einen oder mehrere Befehle enthalten. Die Notation in **Tcl** ist dabei an mancher Stelle angelehnt an UNIX-Notationen, an anderer Stelle findet man des öfteren direkte Bezüge zu der Programmiersprache C, wie z.B. die Operatoren, welche fast komplett aus C übernommen worden sind.

Die Syntax eines **Tcl**-Befehls ist der Name des Befehls, gefolgt von seinen Argumenten, abgetrennt durch Leerzeichen. Semikolons und Zeilenenden, sowie die Klammerung einer Gruppierung, bedeuten in **Tcl** das Ende einer Anweisung:

```
Befehl arg1 arg2 arg3 ...
```

Diese Syntax wird in **Tcl** prinzipiell durchgängig verwendet, wobei lediglich *Gruppierungen* und *Substitutionen* vom Interpreter vor der Ausführung eines solchen Befehls berücksichtigt werden. Mit der Gruppierung ist es möglich, mehrere Worte in einem einzelnen Argument unterzubringen. Die Substitution ermöglicht die Einbindung von Variablen und anderen Befehlsaufrufen in dem Befehl.

Ein Befehl wird dabei in zwei Schritten ausgewertet. Zunächst bricht der Interpreter die Befehle in Worte auf und wertet Substitutionen aus. Das erste Wort wird benutzt, um den Namen der Befehlsprozedur zu bestimmen, die folgenden Worte sind ihre Argumente. Sie kann diese dann beliebig auswerten.

Es gibt einige reservierte Zeichen in **Tcl**, die mit Vorsicht in einem anderem, als dem vorgegebenen Kontext, gebraucht werden sollten. Die Erklärungen folgen in den nächsten Kapiteln. Im Einzelnen wären hier zu nennen:

- Anführungszeichen `"..."` und geschweifte Klammern `{...}` für die Gruppierung
- eckige Klammer `[...]`, das `$`-Zeichen und der Backslash `\` für die Substitution
- das `#`-Zeichen für den Kommentar

Hier sei schon eine weitere, für das Verständnis der Arbeitsweise des **Tcl**-Interpreters grundlegende Eigenschaft angeführt: **Tcl** wertet jedes Zeichen im auszuführenden Befehlsstring nur ein einziges Mal aus.

Jede Eingabezeile wird in **Tcl** gleich behandelt. Klammerungen und ähnliches haben lediglich Bedeutung für die Gruppierung/Substitution, sind aber nicht relevante Elemente der Grammatik und müssen dementsprechend auch nicht unbedingt bei jedem Befehl an derselben Stelle stehen. Der Interpreter wertet stur die Reihenfolge Befehl, Argument 1, Argument 2, usw. aus. Er kennt dabei keine grammatikalischen Zwänge, die viele C Programmierer gerne in die **Tcl**-Skripte hinein interpretieren.

Erste Tcl-Befehle

Zum Arbeiten mit **Tcl** wird man sicher den Befehl *exit*, der ein Verlassen der Shell bewirkt, benötigen. Wenn man sein Programm kommentieren möchte, kann man durch das Zeichen '#' eine Kommentarzeile einleiten. Man kann Kommentare auch durch "... ;# ..." mit dem Semikolon an einen Befehl in der gleichen Zeile anfügen.

Im ersten Beispiel sollen mit dem Befehl *expr*, welcher eine mathematische/ logische Auswertung vornimmt, einige der Operatoren angewendet werden:

Multiplikation:

```
expr 2 * 2

> 4
```

Links Shift:

```
expr 9 << 3

> 72
```

Wie man sieht, liefern alle Befehle ein in der nächsten Zeile ausgegebenes Ergebnis zurück. Dies geschieht grundsätzlich, wenn man aus der Shell heraus Befehle auslöst. So auch bei den folgenden Beispielen, welche jeweils einen logischen Ausdruck auswerten, und 0 für FALSE oder 1 für TRUE zurückgeben:

```
expr (5<6)

> 1

expr (3 == 4) && (5 < 6)

> 0
```

Der Befehl *set* wird dazu benutzt, um Variablen einen Wert zuzuweisen. Dabei kann die Variable Namen aus beliebigen Zeichen annehmen, Groß-/Kleinschreibung wird unterschieden und es ist nicht nötig Variablen vor ihrem ersten Auftreten zu deklarieren:

```
set var_1 5

> 5
```

Der Befehl *puts* gibt einen String aus. Als erstes Argument muss der Ausgabekanal angegeben werden. Hier im Beispiel ist dies der Standardausgabekanal *stdout*. Danach folgen die auszugebenden Argumente:

```
puts stdout Tcl_Seminar

> Tcl_Seminar
```

Kapitel 3: Tcl, die Grundlagen

Auszug aus:
http://sol.ea.rbs.schule.ulm.de/sol/datatech/progspra/tcltk/uniessen/start.htm

Ein **Tcl**-Programm besteht aus Anweisungen, die durch „Newline“ oder ’’;“ getrennt werden. Eine Anweisung besteht aus einem oder mehreren Worten. Das erste Wort ist stets ein Befehl, weitere Worte sind Argumente des Befehls. Dieser Einführung in **Tcl/Tk** liegt **Tcl** Version 7.4 oder höher und **Tk** Version 4.0 oder höher zugrunde.

Variablen

Variablen werden nicht deklariert sondern existieren ab der ersten Zuweisung.
Z.B. ruft die Anweisung *set a 35* die Variable *a* ins Leben und weist ihr den Wert *35* zu.
Zeichenketten werden genauso zugewiesen: *set t ’’Hallo’’*.
Variablen brauchen keinen Typ, da sie implizit alle vom Typ String sind. Dies ist auch der Grund dafür, dass man in **Tcl** arithmetische Ausdrücke nicht unmittelbar hinschreiben kann, sondern die Prozedur *expr* benutzen muss. (Analog der Shell-Programmierung unter UNIX).
Soll eine Variable gelesen werden, dann muss ein *$* davor stehen.

z.B. *set x $a* weist der Variablen *x* den Wert der Variablen *a* zu.

Einfache Variablen

Die grundlegende Datenstruktur von **Tcl** ist der String. Auf Basis der Strings werden einfache Datentypen realisiert, wie

- Integer,
- Floatingpoint Variablen
- Strings

Dabei ist keine explizite Typisierung wie in C notwendig, eine Variable muss – ähnlich wie in den meisten Basic-Dialekten – noch nicht einmal vor dem ersten Auftreten deklariert werden. Das erspart dem Programmierer einerseits einiges an Deklarationsaufwand, kann aber zu großen Problemen bei der Programmierung und insbesondere bei der Fehlersuche führen. Man muss deshalb ständig darauf achten, was man gerade in einer Variable gespeichert hat, weil sonst Prozeduren, die z.B. einen Integerwert erwarten bei einem anderen Datentyp als Übergabe entweder unsinnige Rückgaben erzeugen oder aber zu Fehlermeldungen führen.

Auf Basis der Strings finden zwei umfangreichere Datenstrukturen Unterstützung in der Basissprache: Die Liste und das Array.

set: Initialisieren einer Variablen

Die einfachen Variablen werden, wie bereits gesehen, mit dem *set* Befehl elementar verwaltet. Es sind die Operationen: Initialisierung, Neubelegung und Auslesen möglich. Dabei wird folgende Syntax verwandt:

```
set varName ?value?
```

Wird das optionale *value* erstmals mit *set varName value* angegeben, so wird *varName* mit dessen Wert initialisiert. z.B.:

```
set a 5

> 5
```

Wenn man nun denselben *varNamen* abermals als Argument an set *varName value* übergibt, so wird eine Neubelegung vorgenommen.

```
set a 10

> 10
```

Gibt man *value* nicht an, so wird die Variable ausgelesen:

```
set a

> 10
```

Ebenso wie die hier verwendeten Integer-Belegungen könnten auch genauso gut Strings oder Floating Point Variablen initialisiert werden, wie z.B.:

```
set b Tcl_Network

> Tcl_Network

set c 1.5

> 1.5
```

unset: Löschen einer Variablen aus dem Speicher

```
unset name ?name name ...?
```

Die Anweisung *unset* ermöglicht das Löschen einer / mehrerer Variablen aus dem Speicher. Nach *unset* wird der Bezeichner nicht mehr vom Interpreter erkannt. Bei Arrays wird nur das spezifizierte Element gelöscht. Bei einem Fehler wird ab dem Fehler *unset* nicht weiter durchgeführt.

incr: Addieren bzw. Variable um 1 erhöhen

```
incr var ?increment?
```

Die Anweisung *incr* ist eine einfache Möglichkeit, neben *set*, Werte von Variable einfach zu verändern. Es addiert *increment* auf die Variable var. Wird *increment* nicht angegeben, so wird einfach 1 addiert. Bsp.:

```
incr a -5

> 5
```

Substitution

Es gibt drei Arten von Substitution:

- Variablensubstitution
- Befehlssubstitution
- Backslash-Substitution

Um den Wert einer Variablen auszulesen, kann man die so genannte Variablen- Substitution nutzen. Sie nutzt das '$'-Zeichen, um zu signalisieren, dass der Wert und nicht die Adresse der Variable zurückgegeben werden soll.

Bsp.:

```
set var_1 5

> 5

expr $var_1 + 5

> 10
```

Möchte man gerne das Ergebnis einer Operation an eine andere übergeben, so nutzt man am einfachsten die Befehlssubstitution. Diese Form der Ersetzung nutzt die eckigen Klammern '[ ]', um den Rückgabewert einer Operation in eine andere einzufügen:

```
set var_2 [expr $var_1 *4]

> 20
```

Die letzte Form der Substitution ist die Backslash-Substitution, die den Backslash '\' nutzt, um einerseits vom Compiler benutzte Zeichen, wie das '$' Zeichen in anderem Kontext benutzen zu können. Andererseits können Zeichencodes, die eine bestimmte Bedeutung haben, und Zahlen in anderen Systemen als dem dezimalen an den Interpreter übergeben werden: So übergibt

```
set backslash \\

> \
```

der Variablen *backslash* das reservierte Zeichen '\'. Der Befehl

```
set zeilenschaltung \14
```

übergibt der Variable *zeilenschaltung* den dezimalen Wert des Zeichencodes für die Zeilenschaltung. Synonym hätte man auch verwenden können:

```
set zeilenschaltung \n
```

Selbstverständlich kann Substitution auch *kaskadierend* erfolgen, das heißt eine beliebige Schachtelung der Substitutionen ist möglich, wie

z.B.:

```
set var_1 5

> 5

set var_2 var_1

> var_1
```

Nun hat `var_2` als Wert den String "var_1", nicht den Wert von `var_1`.

```
set x [set [set var_2]]

> 5
```

In der geschachtelten Substitution wird zunächst *[set var]* durch den String *var_1* ersetzt. Nun übernimmt das umgebende *set* diesen Wert als Argument, um dann mit *set var_1* der Variable *x* im äußersten *set* die *5* zu zuweisen. Man kann solche Verschachtelungen beliebig tief fortsetzen.

Strukturierte Variablen

Strukturierte Variablen gibt es in **Tcl** eigentlich gar nicht. Es gibt aber so etwas wie Arrays. Die Verwendung ist ziemlich einfach, da man sie einfach benutzen und auch alles als Indizes verwenden kann.

set tage(Januar) 31
set tage(Februar) 28
set tage(Maerz) 31
set tage(April) 30

Man kann aber ebenso gut Zahlen als Indizes verwenden oder Zahlen und Text mischen.

Gruppierung

Es gibt zwei Möglichkeiten der Gruppierung, die es ermöglichen soll mehrere Worte in einem einzigen Argument zusammenzufassen und als ein einziges Wort zu interpretieren. Die erste Möglichkeit sind die Hochkommata, welche innerhalb des Wortes die Anwendung der drei oben genannten Varianten der Substitution zulässt. Um genau dies zu unterbinden, kann man anstatt dessen die geschwungenen Klammern benutzen. Bsp.:

Mit Substitution:

```
set mit_subst "Substitution im Wort"

> Substitution im Wort
```

Ohne Substitution:

```
set ohne_subst {Keine Substitution im Wort}

> Keine Substitution im Wort
```

Grundsätzlich muss hier noch bemerkt werden, dass die Hierarchie lautet: Gruppierung geht vor Substitution. Im Buch von Brent B. Welch [Welch95[10]] wird hierzu auf Seite 10 ein anschauliches Beispiel, mit dem Titel: *Example 1-13: Embedded command and variable substitution,* gegeben, welches zeigt, wie schnell hier ungewollte Fehler entstehen können:

```
set x 7 ; set y 9

> 9

puts stdout $x+$y=[expr $x + $y]

> 7+9=16
```

Hier geht der Interpreter so vor, dass er zunächst im ersten Durchlauf die Substitution des Ausdrucks $x+$y in 7+9 vornimmt, dann auf die Klammer stößt, und sich selbst rekursiv wieder aufruft. Dort ersetzt er wieder erst $x und $y, und kommt dann zur Auswertung von `expr`, das dann 16 zurückliefert. Daher bekommt *`puts`* als zweites Argument den String "7+9=16" anstatt einer vielleicht gewollten Zahl zurück.

Listen

Listen haben in **Tcl** die Form *{ Oktober November Dezember }*.
Man kann einer Variablen eine Liste zuweisen:

[10] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Welch95

set l { Oktober November Dezember }

Auf ein bestimmtes Listenelement kann man mit der Prozedur *lindex* zugreifen.
Das erste Listenelement hat den Index 0.
Also gilt
z.B.:
set el1 [lindex $l 1]

Hier wird der Variablen *el1* das Element mit dem Index 1, also *November* zugewiesen. Listen können selbst Listen als Elemente enthalten. Es gibt auch Prozeduren, die mit Listen arbeiten, oder solche, die ihre Argumente zu einer Liste zusammengefaßt haben.

concat Zusammenfassen von Listen :

set a 8
set ll [concat a b $a "Hans Meier" { x yy Z }]
Man beachte, dass die neue Liste nicht 5 Elemente enthält, sondern 8.

ll beinhaltet die Liste *{ a b 8 Hans Meier x yy Z }*.

Genaueres zur Liste

Entgegen der Implementierung von Listen in anderen Programmiersprachen sind Listen in **Tcl** nur Strings, die entsprechend interpretiert werden. Es gibt eine ganze Reihe von Befehlen zur Manipulation von Listenstrukturen, die zunächst in ihrer Syntax tabellarisch zusammengefasst werden, und dann an einem kurzen Beispiel kurz erläutert werden sollen. Man wird dabei das doch recht einfach zu verstehende Listenkonzept von **Tcl** schnell durchschauen.

Die Listenbefehle von Tcl	
list a1 a2 ... an	Liste erstellen aus den Argumenten a1 ... an
lindex list i	Gibt das i-te Element von list zurück, angefangen mit 0
llength list	Länge der Liste
lrange list i j	Gibt die i -ten bis j -ten Elemente von list zurück
lappend listVar a1 a2 ...an	Hängt die Argumente a1..an an list an
linsert list index a1 a2 ...an	Fügt a1..an ab index in die Liste list ein
lreplace list i j a1 a2 ... an	Ersetzt i bis j durch a1..an. Die Anzahl muss dabei stimmen.
lsearch mode list value	Liefert den Index, der value entspricht oder -1 für kein Ergebnis, mit dem mode: -exact (Zeichen für Zeichen), -glob (default-wie UNIX-Shell), -regexp(umfangreiche Prüfung); siehe String Routine für Bedeutung
lsort switches	Sortiert Liste mit Optionen switches: -ascii, -integer , - real , -increasing, -

list	decreasing ; -command command;
concat a1 a2 a3 ..	Mehrere Listen in eine zusammenfügen
join list joinString	Macht aus einer Liste einen String mit Trennstring joinString
Split string splitChars	Macht aus einem String eine Liste mit Trennung splitChars

Das Schlüsselwort *end* gibt grundsätzlich das Ende der Liste an, und kann dementsprechend für die Argumente oben eingesetzt werden. Der Anfang ist immer 0.

Die Konstruktion einer Liste geschieht beispielsweise durch:

```
set x {rot gelb}

> rot gelb
```

Die Liste hat nun zwei Elemente: *rot* und *gelb*. Man sieht, man kann Listen auch nur mit *set* deklarieren. Man sollte jedoch das *list*-Kommando benutzen, weil es spezielle Charaktere (wie z.B. $) so behandelt, dass sie trotzdem als ein Element behandelt werden können:

```
set y [list gruen gelb \$ $x]

> gruen gelb {$} {rot gelb}
```

In der Ausgabe bedeuten Klammerungen dabei, dass es sich um ein einziges Element handelt. Die hier konstruierte Liste hat vier Elemente. Mit *lappend* kann man neue Elemente an bestehende Listen anfügen. Gibt es die angegebene Listenbezeichnung noch nicht, so wird genau wie bei *list* eine neue Liste erzeugt.

```
lappend y {violett grau} blau

> gruen gelb {$} {rot gelb} {violett grau} blau
```

Es wurden nun zwei Elemente angefügt, die Liste hat nun die Länge 6. Mit *llength* lässt diese sich ermitteln.

```
llength $y

> 6
```

Um auf einzelne Listenelemente zuzugreifen, etwa um einen Durchlauf zu realisieren, dient der Befehl *lindex*, der aus einer Liste ein spezifiziertes Element zurückgibt. Dabei wird mit der Null zu zählen begonnen.

```
lindex $y 0

> gruen
```

Möchte man nicht nur ein Element, sondern eine ganze Teilliste zurückgeliefert bekommen, so ist der Befehl *lrange* hilfreich. Er erwartet wiederum die Liste und das Startelement, sowie zusätzlich das Endelement, als Argumente. Er liefert als Ergebnis eine Liste zurück.

```
lrange $y 2 end

> {$} {rot gelb} {violett grau} blau
```

linsert dient dem Einfügen in eine Liste ab einem spezifizierten Element. Alle darauf folgenden Elemente werden nacheinander in die Liste eingefügt.

```
linsert $y 1 neu1 neu2 neu3

> gruen neu1 neu2 neu3 gelb {$} {rot gelb} {violett grau} blau
```

Möchte man zwei oder mehrere ganze Listen zusammenfügen, so nutzt man ***concat***, das selbst wiederum eine neue Liste zurückliefert.

```
set a [concat {1 2 3} {4 5 6} {7 8 9}]

> 1 2 3 4 5 6 7 8 9
```

lreplace ersetzt die angegebenen Bereich an Elemente durch eine gleiche Anzahl neuer Elemente und liefert eine neue Liste zurück.

```
lreplace $a 0 2 a b c

> a b c 4 5 6 7 8 9
```

lsearch gibt den Index des zu suchenden Elementes zurück.

```
lsearch {a b c d e f} c*

> 2
```

Zum sortieren von Listen dient *lsort*. Default ist die Sortieroption *-ascii, -increasing*. Als command kann eine eigene Sortier-Prozedur angegeben werden, die bestimmt auf welche Weise zwei Strings miteinander verglichen werden sollen. Sie muss zwei Argumente haben, in denen die beiden Strings von *lsort* übergeben werden. Ihr Ergebnis muss das Resultat einer string compare Operation sein, also 0, wenn gleich, -1, wenn arg1 vor arg2, sonst 1. (Bei Unklarheiten findet man hierzu ein Beispiel im Welch95 auf Seite 35)

```
lsort {b c a f e d}

> a b c d e f
```

Zwei besondere Befehle sind *split* und *join*. Sie beschäftigen sich mit der Umsetzung eines Strings in eine Liste (*split*), und umgekehrt (*join*). Dabei müssen jeweils ein/mehrere Charaktere angegeben werden, die die einzelnen Listenelemente im String voneinander trennen.

```
split {elt1/elt2/elt3/elt4} /
```

```
> elt1 elt2 elt3 elt4

join {elt1 {elt2_1 elt2_2} elt3} /

> elt1/elt2_1 elt2_2/elt3
```

Arrays

Arrays sind in **Tcl** direkt verfügbar, und sind wiederum über Strings implementiert. Es gibt ein- und mehrdimensionale Arrays, wobei mehrdimensionale Arrays nicht tatsächlich implementiert werden, aber leicht zu simulieren sind. Dabei unterscheiden sich die Arrays in **Tcl** gegenüber Feldern in anderen Sprachen neben ihrer internen Repräsentation, auch durch eine Eigenschaft, die Ousterhout in seinem Buch auf Seite 49 als *assoziativ* bezeichnet. Damit ist gemeint, dass nicht nur Integerwerte, sondern auch Strings als Indizes für Feldelemente genutzt werden können. Welch spricht in seiner Einleitung zu den Arrays (S.35) von einem "mapping from strings to strings". Intern sind Arrays als Hash-Tabelle angelegt, so dass ein Suchen eines beliebigen Elementes ähnlich lange dauern sollte.

Array-Variablen werden wie die einfachen Variablen mit *`set`* definiert, wobei es nicht notwendig ist, das ganze Array auf einmal zu deklarieren:

```
set arr(index) ?value?
```

`arr` ist dabei der Name des Arrays, *`index`* der Index-String und `value` eine Belegung der Variablen.

Ganze Arrays können durch Schleifen initialisiert werden. Im Abschnitt zur For-Schleife ein solches Beispiel angegeben.

Für die Assoziativität von Arrays führt Ousterhout auf Seite 49 seines Buches folgendes einführendes Beispiel an:

```
set Umsatz(Januar) 87966

> 87966

set Umsatz(Februar) 95400

> 95400

set Umsatz(Januar)

> 87966
```

Die grundlegenden Array-Operationen dürften damit angesprochen sein. Wie bereits erwähnt ist es nicht direkt möglich, mehrdimensionale Arrays in **Tcl** zu behandeln. Doch durch die Assoziativität ist es sehr einfach diese zu simulieren.

```
set Zwei_Dim(1,1) 11
```

```
> 11

set Zwei_Dim(2,2) 34

> 34
```

Tatsächlich ist hier nicht ein zwei dimensionales Array definiert, sondern ein eindimensionales, welches über die Strings "(1,1)" und "(2,2)" referenziert wird. Ansonsten bemerkt man aber in der Behandlung keine Nachteile, so dass man so recht einfach mit mehrdimensionalen Feldern arbeiten kann. Allerdings sollte man darauf achten, keine Leerzeichen in die Referenzierungsstrings einzufügen, wie z.B. "(1 , 1)", wenn man dies nicht auch in der Deklaration getan hat, da das entsprechende Element sonst nicht gefunden werden kann.

Neben diesen elementaren Operationen auf Arrays sind auch hier wieder Komplexere definiert. Dabei werden die Befehle in dem Kommando *array* gebündelt, das dann durch Subkommandos als erstes Argument in der tatsächlichen Funktion gesteuert wird. Dies ist eine übliche Verfahrensweise in **Tcl**, und wird auch ähnlich in dem später betrachteten **OTcl** angewandt, um die Methoden, usw. von Objekten/Klassen anzusprechen.

Die Array Kommandos geben Informationen über ein Array aus. Diese sind recht deutlich und leicht verständlich, so dass sie in einer Tabelle zusammengefasst und kurz erläutert werden sollen:

Der Befehl array	
Array exists arr	1 (true), falls arr eine ArrayVar ist
Array get arr	Gibt eine Liste zurück, die für alle Elemente der Arrays abwechselnd den index und den zugehörigen Wert enthält
array names arr ?pattern?	Liste aller Indizes, die für arr definiert sind oder, falls angegeben, die auf pattern passen. Kann als Argument einer foreach Schleife zum Durchlauf genutzt werden
Array set arr list	Initialisiert Array mit aus einer Liste list heraus, die abwechselnd einen Index und eine Value enthält
Array size arr	Anzahl der Indizes in arr
array startsearch arr	Gibt token für Suchvorgang durch arr im korrespondierenden Hash-Table zurück
array nextelement arr id	Gibt nächste Element in der Suche nach token id zurück, bzw. einen leeren String, falls keine weiteren Elemente vorhanden sind.
array anymore arr id	1, falls noch Elemente in der Suche vorhanden sind
array donesearch arr id	Suche mit token id beenden

Termersetzung

Termersetzung tritt z.B. dann auf, wenn man das Ergebnis eines Ausdrucks zuweisen oder anders verwenden will. Es seinen dazu zunächst einige Beispiele aufgeführt:

(1) set a 5
*(2) set b [expr $a * 5]*
(3) set c $a
(4) set d { $a }
(5) set e "Die Variable hat den Wert $a"

Was bedeutet das nun alles?

In Bsp. (1) wird der Variablen der Wert 5 zugewiesen. Hier ist zu beachten, dass die Prozedur *set* zwei Argumente erwartet (hier a und 5).

Auch in Bsp. (2) hat *set* zwei Argumente, nämlich *b* und *[expr $a * 5]*. Dieser Ausdruck wird zuerst ausgewertet und dann wird das Ergebnis der Variablen *b* zugewiesen. *b* hat also anschließend den Wert *25*. Zwischen den eckigen Klammern muss ein Kommando (also ein Prozeduraufruf) stehen. Alle Kommandos haben einen Rückgabewert. Dieser Rückgabewert wird vom Aufrufendem benutzt.

In Bsp. (3) wird der Wert der Variablen a gelesen und der Variablen c zugewiesen.

Bsp. (4) weist der Variablen *d* genau das zu, was zwischen den geschweiften Klammern steht, einschließlich der vorkommenden Blanks. *d* hat also den Wert „$a“. Die geschweiften Klammern bewirken also, dass die **Tcl**-Shell das, was dazwischen steht nicht auswertet.

In Bsp. (5) wird der Variablen *e* eine Zeichenkette zugewiesen, die eine Variable enthält. Diese Variable wird vor der Zuweisung ausgewertet, wodurch *e* den Wert „Die Variable hat den Wert 5“ erhält.

Numerische Ausdrücke

Numerische Ausdrücke können in Bedingungen wie *if* und *while* vorkommen und bei expliziter Termauswertung mit *expr*.

Beispiele:

expr 2 + 2

if { (2 + 3) > 0 } {
puts "TRUE"
}

Kontrollstrukturen

Kontrollstrukturen werden (wie auch die Prozeduren) als eine einzelne Anweisung interpretiert. Sie haben in der Regel einen Rumpf, der sich über mehrere Zeilen (geklammert) erstrecken kann, aber als ein einziges Argument interpretiert wird. Vor diesem Rumpf stehen in der Regel eine Reihe von Argumenten, die der Auswertung von Abbruchbedingungen, Auswertungen und ähnlichem dienen. Schleifen werden durch die Anweisungen *while*, *for* und *foreach* realisiert, Programmverzweigungen übernehmen *if* und *switch*, *catch* dient der Fehlerbehandlung, *break*, *continue*, *return* und *error* übernehmen die interne Kontrolle der Strukturen. Hier nun die Syntax der verschiedenen Strukturen, und eine kurze Erläuterung:

For-Schleife:

```
for initial test final body
```

In *initial* stehen einleitende Initialisierungen der Variable für die For-Schleife, *test* bezeichnet die Abbruchbedingung, und *final* bezeichnet eine Anweisung, die nach dem Ende jedes Schleifendurchlauf dem Rumpf angehängt wird, etwa zum heraufsetzen der Schleifenvariablen. Der *body* ist das Argument, welches die eigentliche Schleife enthält.

Wie im Abschnitt über Arrays angesprochen, soll nun im Beispiel ein ganzes Array mit den Werten 1*2*...*x für die Array-Elemente arr(x) initialisiert werden:

```
set arr(0) 1

for {set i 1} {$i <= 10} {incr i} {

set arr($i) [expr $i * $arr([expr $i-1])]

}

set arr(10)
>3628800
```

Foreach-Schleife:

```
foreach loopVar valueList body
```

Die *Foreach*-Schleife ist ein spezielles **Tcl**-Konstrukt, welches es erlaubt mit einer Schleifenvariable eine vorher festgesetzte Liste von Werten zu durchlaufen und jeweils den *body* einmal auszuführen. Diese Variable wird durch *loopVar* festgelegt, die Liste wird in *valueList* festgelegt. Bspw. könnte das in der Form {1 7 10 9 5} geschehen. Auch hier ein Beispiel, das die Liste der ungeraden Ziffern in einer Variable *v1* aufsummiert:

```
foreach i {1 3 5 7 9} {

set v1 [expr $var+$i]

}
```

While-Schleife:

```
while expression body
```

Hier ist *expression* ein boolscher Ausdruck, wie zum Beispiel *{$var > 10}*. *body* bezeichnet den Körper der Anweisung. Möglicherweise steht hier ein einzelner Befehl, oder aber eine gruppierende Klammerung. Hier das Beispiel der *For*-Schleife mit einer *While* Schleife umgesetzt:

```
set arr(0) 1

set i 1

while {$i <= 10} {

set arr($i) [expr $i * $arr([expr $i-1])]

incr i

}

set arr(5)

>120
```

If-Anweisung:

```
if expression then body1 else body2
```

Eine gewöhnliche *If*-Bedingung mit der ein boolscher Ausdruck *expression* ausgewertet wird, und je nach Ergebnis bei *True* der *body1*, sonst der *body2* ausgeführt wird. Ein Beispiel prüft, ob eine Variable Kontostand kleiner Null ist:

```
set kontostand 1000

if {$kontostand < 0} then {

puts stdout "Konto leer"

} else {

puts stdout "Konto ok"

}
```

Switch-Anweisung:

```
switch flags value p1 b1 p2 b2 p3 b3 .... pn bn
```

Die *Switch*-Anweisung entspricht der gleichnamigen C-Anweisung in ihrer Bedeutung, und wertet über den Wert *value*, der bspw. eine Variablensubstitution enthalten kann, die Patterns *p1 .. pn* aus. Beim ersten, der mit *value* im Wert übereinstimmt, wird der zugehörige Body *b1 .. bn* ausgeführt. Danach wird der Befehl abgebrochen. Nur beim letzten Paar *pn bn*

kann man den Wert *default* für *pn* einsetzen, was bewirkt, dass der Body *bn* dann ausgeführt wird, wenn kein anderer Fall zugetroffen hat. *flags* dient der Auswahl der Art der Auswertung. Es ist möglich mehrere Flags hintereinander anzugeben, wobei *-exact* der Default-Wert ist und besagt, dass jeder Pattern-Ausdruck exakt ausgewertet wird. *-glob* ermöglicht die Vergleichsvariante, wie sie von den meisten UNIX-Shells angewendet wird (*, ? und [] werden als Ersatzcharaktere behandelt), *-regexp* benutzt die Vergleichsvariante *regular expression*, -- bedeutet, dass keine Flags gesetzt werden bzw. steht am Ende der Liste aus Flags (nützlich, falls value mit '-' beginnt). Eine weitergehende Erläuterung findet sich bei der Stringauswertung.

Break und continue:

Stößt der Interpreter in einer Schleife auf *break* so bricht er die Schleife ganz ab, stößt er auf *continue*, so setzt er mit der nächsten Iteration fort.

Catch:

catch *command ?resultVar?*

Catch prüft eine Anweisung *command* auf Fehler, und sorgt dafür dass bei Fehlern nicht direkt eine Meldung angezeigt wird. Liegen Fehler im Code oder in Benutzung vor, so wird der zugehörige Fehlercode zurückgeliefert. Anderenfalls ist der Rückgabewert 0. Der Code ist der Rückgabewert der Catch-Anweisung, und wird für weitere Behandlung in *resultVar* eingetragen. Er kann dementsprechend als Boolscher Ausdruck ausgewertet werden, wie in der *If*-Anweisung im folgenden Beispiel:

```
if [catch {

....commands ....

} result] {

global errorInfo

puts stderr $result

puts stderr $errorInfo

} else {

puts stout ok

}
```

Die variable *errorInfo* wird hier vom Interpreter gesetzt, und gibt die Position im Stack aus Sicht des Fehlers an.

Error:

error message ?info? ?code?

Error erzeugt einen Fehler mit der Ausgabe *message*, der das Skript ohne ein umgebende *catch*-Anweisung zum Abbruch bringt, dabei dient *info* der Rettung der Fehlerinformation, wie sie in *errorinfo* steht und der *code* bestimmt den maschinenlesbaren Fehlercode.

Prozeduren

Prozeduren werden in **Tcl** nach der folgenden Syntax zusammengesetzt:

```
proc name arglist body
```

Hier ist *proc* ein Befehl, der den Beginn der Prozedur bestimmt, *name* ist der Bezeichner für die Prozedur, *arglist* sind die übergebenen Parameter und *body* ist die eigentliche Ausformulierung der Prozedur. Alles ist wiederum ein einziger Befehl mit mehreren, unter Umständen gruppierten Argumenten. Hier ein Beispiel:

```
proc berechne {a b} {

set c [expr $a + [expr $b * 2]]

return $c

}

berechne 4 5
>14
```

Hier gruppieren die Klammern jeweils das zweite und das dritte Argument zu einem Wort. Ebenso gut wäre eine Lösung mit Hochkommas denkbar. **Tcl** gibt bei Prozeduren grundsätzlich den Wert des letzten Befehls zurück, das `return-` Kommando ist hier optional. **Tcl**-Prozeduren sind also prinzipiell immer Funktionen.

Die komplette Syntax des *return*-Kommandos lautet:

```
return ?code? ?errorinfo? ?-errorcode? String
```

Wobei *code* entweder *ok*, *error*, *break*, *continue* oder ein Integercode ist. Falls der code *error* ist, so können mit *errorinfo* und *errorcode* die entsprechenden Werte, wie bei *error* beschrieben gesetzt werden. `String` ist ein allgemein ausgegebener Rückgabewert, wie er auch oben gesetzt wurde, und der auch ganz alleine zurückgeschickt werden kann.

Zum Scope (Sichtbarkeits- und Gültigkeitsbereich)

Variablen- und Prozedurnamen sind in **Tcl** per default lokal, d.h. jede Prozedur hat ihren eigenen Sichtbereich, in dem Variablen- und Prozedurnamen beliebig benannt werden können, ohne mit den Variablen anderer prozeduraler Ebenen in Konflikt zu geraten. Das Kommando

```
global var1 var2 var3 ...
```

macht alle angegebenen Variablen global verfügbar.

```
upvar ?level? varname localvar
```

Upvar verknüpft die lokale Variable *localvar* mit einer Variable *varname*, die in der Hierarchie höher liegt, wobei *level* optional angibt, wie viele Ebenen diese höher liegt (default ist 1; 0 bedeutet global).

eval: Indirekte Ausführung eines Kommandos

Der Befehl *eval* dient zur indirekten Ausführung eines Kommandos. Wenn man einen Befehl dynamisch konstruieren möchte, um ihn später aufzurufen, benutzt man die *eval*-Anweisung, die die angegeben Argumente evaluiert und damit den **Tcl**-Interpreter aufruft. Bsp.: Will man

```
puts stdout "eval Bsp1"
```

später ausführen, so kann man den eval-Befehl wie folgt einsetzen:

```
set cmd {puts stdout "eval Bsp1"}

.....

eval $cmd

> eval Bsp1
```

Will man eine dynamisch veränderbare (und somit auch mit *unset* auflösbare) Struktur in der aktuellen Belegung später in einem Kommando aufrufen, so sollte man nicht ähnlich wie folgt aufrufen:

```
set s "eval Bsp2"

set cmd {puts stdout $s}

...

eval $cmd
```

Hier könnte in dem Zwischenbereich ein *unset s* stehen, das zu einer Fehlermeldung führt, oder aber ein *set s ...* , das das *s* mit einem anderen Wert belegt. Stattdessen bietet sich hier an,

```
set cmd [list puts stdout $s]
```

zu benutzen, denn beim *list*-Kommando wird die Substitution vor der Ausführung durchgeführt, was dafür sorgt, daß ein späteres

```
eval $cmd
```

den Wert *$s* und nicht die Referenz als drittes Argument übergeben bekommt. Auch das Nutzen von Hochkommata als Gruppierung kann zu Fehlern führen, da Leerzeichen dann unter Umständen als Abtrennungssymbol interpretiert werden:

```
set cmd "puts stdout $s"
```

würde bspw. eine Fehlermeldung liefern, die daraus resultiert, dass "eval" und "Bsp2" nun zwei eigenständige Argumente zu sein scheinen. Diese möglichen Schwierigkeiten mit *eval* resultieren daraus, dass intern zunächst, genau wie es der *concat*-Befehl macht, eine einzige Liste aus allen Argumenten von *eval* erzeugt wird. Hier wären im obigen Beispiel "eval" und "Bsp2" zwei eigenständige Listenelemente. Die Technik, das List-Komanndo anzuwenden, vermeidet solche Probleme und sollte deshalb bei allen dynamischen Strukturen, sofern sie einem Kommando übergeben werden, dass im nachhinein ihre vorher intern verbundenen Argumente auswertet, benutzt werden.

Prozesse

Es gibt in **Tcl** eine Reihe von Befehlen zur Bearbeitung von Prozessen. Prozesse können mit den beiden Kommandos *exec* und *open* erzeugt werden. *open* wird im nächsten Unterpunkt beschrieben.

exec: Ausführen von Befehlen (siehe VL Betriebssysteme)

Das Kommando *exec* wird benutzt, um UNIX-Befehle auszuführen, mit der Syntax:

```
exec cmd
```

Dabei wird einer oder mehrere neue Prozesse erzeugt. Die **Tcl**-Shell versucht schon von sich aus automatisch Befehle, die nicht als **Tcl**-Befehle erkannt werden, als UNIX-Befehle zu interpretieren. Dies kann aber durch das setzen von *auto_noexec* unterdrückt werden.

Bsp.:

```
set auto_noexec anything
```

Nun sind alle automatischen UNIX-Interpretationen ausgeschaltet. Mit *exec* kann man die Shell zwingen, ein Kommando als UNIX-Befehl zu interpretieren.

Ein Beispiel für das *exec* Kommando ist:

```
set datum [exec date]
```

Das Systemdatum wird in die Variable *datum* geschrieben.

Prozessumleitung

Jeder Prozeß unter UNIX verfügt normalerweise über drei zugehörige Ströme (streams) : Standardausgabe, Standardeingabe und Standardfehler. Diese werden mit den Funktionen zur File-I/O gelesen und geschrieben. *exec* (wie auch *open*) beherrscht die volle *I/O redirection* und *pipeline* Syntax von UNIX. Mit der *I/O redirection* ist es möglich, die drei Ströme in Files oder in mit *open* geöffnete Ströme umzuleiten. Eine *pipeline* ist ein standardisiertes Konstrukt, das die Standardausgaben eines Befehls an die Standardeingaben eines anderen Befehls anhängt. Es können beliebig viele Programme miteinander verbunden werden, und somit mehrere Prozesse mit einem Befehl erzeugt werden, wobei die Ausgaben auf eine vorher bestimmte Art und Weise an die Eingabe des nächsten Pipelineelementes übergeben werden.

Eine komplette Auflistung der Möglichkeiten zur Umleitung/*pipeline* findet sich bei Welch auf Seite 64 [Welch95[11]], im Buch von Ousterhout auf Seite 123 [Ousterhout94[12]]). Hier sei nur erwähnt, dass "prozess_1 | prozess_2" die einfachste, direkte Übergabe von Ausgaben realisiert.

Beendigung von Prozessen:

```
exit ?exitStatus?
```

exit beendet den aktuellen Prozess, den das **Tcl**-Skript ausführt. *exitStatus* ist eine optional anzugebende Integerzahl, die den Status des Ausstiegs charakterisiert. *exit* selbst hat keine Rückgabe, da er ja den Prozess beendet.

Arbeiten mit Dateien

Das File-I/O System von **Tcl** basiert auf UNIX. Die Befehle setzen einen Kernel voraus, der dem POSIX-Standard folgt. Das Filesystem stellt eigentlich alle Befehle zur Verfügung, welche die C-Library auch anbietet. Die Befehlsnamen ähneln zum Teil den C-Befehlen. Es werden dabei die Dateinamen genauso, wie bei UNIX üblich, verwendet. Es wird vom aktuellen Verzeichnis ausgegangen, und dann mit dem Slash "/" in die Tiefe verzweigt.

Die Datei-Funktionen öffnen jeweils einen der Ströme, nehmen ihre Operationen vor und schließen ihn dann wieder. Dies ist eine in den meisten Programmiersprachen übliche Vorgehensweise. Wichtig ist, dass die jeweiligen Standardströme (je nach Flags) geöffnet werden, und das die Ausgabe auch über diese erfolgt. Beispielsweise sollte man daher Fehlermeldungen auch über den Fehlerstrom *stderr* ausgeben.

[11] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Welch95
[12] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm#Ousterhout94

Öffnen von Dateien:

```
open what ?access? ?permissions?
```

Öffnet eine Datei, und liefert die *stream*-ID (wird für alle folgenden Operationen benötigt) eines Files oder einer Pipeline zurück. *what* ist der Name des Files oder aber eine Pipelinebeschreibung, wie sie in *exec* genutzt wurde. Optional kann man folgende *access*-Argumente geben:

- r: nur lesen, File muss existieren
- r+: lesen und schreiben, File muss existieren
- w: nur schreiben. File wird überschrieben oder neu erzeugt
- w+: lesen und schreiben, File wird überschrieben oder neu erzeugt
- a: nur schreiben, File muss existieren und Daten werden angehängt
- a+: lesen und schreiben, File muss existieren und Daten werden angehängt

Man kann statt diesen Argumenten, die kompatibel zu denen des *fopen*-C-Library-Komanndos sind, auch die entsprechenden POSIX - Flags verwenden. Das Argument *permission* setzt die *permission*-Bits, *default* ist hier 0666. Weitere Informationen findet man in UNIX -Manuals unter *chmod*.

Schließen von Dateien:

```
close stream_ID
```

Schließt den durch *stream*_ID bestimmten Stream. Falls close vergessen wird, so wird am Ende des Prozesses automatisch geschlossen, was aber natürlich während des Prozesses zur Verschwendung von Ressourcen (evtl. auch zur Überlastung des Betriebssystems) führen kann. Wichtig: Auch *close* kann Fehler erzeugen, bspw. wenn ein Prozess (in einer Pipeline) nicht mit Null, sondern mit einer Fehlermeldung abschließt. Ein Öffnen/Schließen sollte daher mit einer Fehlerbehandlung durch das Kommando catch, welches unter Kontrollstrukturen beschrieben wurde, durchgeführt werden.

Ein- und Ausgabe in Strömen:

```
puts ?-nonewline? ?stream_ID? String
```

Schreibt einen String in den durch *stream*_ID charakterisierten Strom. Ein angegebenes -*nonewline* unterdrückt dabei "\n". Bsp.:

```
puts $fileId "Tcl"
```

Der Befehl *gets* liest einen String aus dem Strom mit der *stream*_ID.

```
gets stream_ID ?varname?
```

Falls nur ein Argument angegeben wird, so wird das Ergebnis zurückgegeben, werden zwei Argumente angegeben wird das Ergebnis im zweiten gespeichert und der Rückgabewert ist die Länge der gelesenen Zeile in Bytes.

Beispiel.:

Variante 1:

```
set a [gets stdin]
```

Variante 2:

```
while {[gets stdin a] == 0} {

# Meldung, daß Eingabe erforderlich ist

}
```

read liest ganze Blöcke von Daten aus dem Stream, die entweder mit *numbytes* in der Länge bestimmt werden oder aber es wird das ganze File auf einmal gelesen. Wird -*nonewline* angegeben, so wird ohne "\n" gelesen:

```
read ?-nonewline? stream ?numbytes?
```

Die Ausgabe durch *puts* erfolgt default-mäßig gepuffert, d.h. es wird erst gewartet, bis der I/O-Puffer voll geschrieben ist, und erst dann wird in die Datei geschrieben. Das Kommando *flush* sorgt dafür, daß sofort physikalisch geschrieben wird.

```
flush stream
```

Schreibt den ganzen Puffer von *stream* sofort.

Zugriff auf Datei – Suchoffset

Die Ein- und Ausgabe erfolgt jeweils ab der aktuellen Position in der Datei. Ändert man diese nicht manuell, so erfolgt der Dateizugriff sequentiell. Allerdings ist es möglich diese Position, die mitunter als Suchoffset bezeichnet wird, zu verschieben. Dadurch ist ein so genannter wahlfreier Zugriff auf Dateien möglich. Die zugehörigen Anweisungen sind:

```
tell stream
```

Liefert den aktuellen Suchoffset

```
seek stream offset ?origin?
```

Setzen des Suchoffsets. Ohne *origin* wird einfach nach dem Element *offset* weiter geschrieben/gelesen. Bsp.:

```
seek $file 100
```

origin ist, wenn genutzt, eines der Schlüsselwörter `start` (Anfang der Datei), *current* (Aktuelle Position) oder `end` (Ende der Datei). Dadurch wird mit dem *offset* relativ zum *origin* gearbeitet.

Beispiel.:

```
seek $file -100 end
```

Mit den beiden Befehlen *tell* und *seek* stehen unter **Tcl** random-access-Dateizugriffe zur Verfügung.

```
eof stream
```

Abfrage des *end-of-file* Status, Null falls beim letzten Lesezugriff noch nicht das Ende der Datei erreicht worden ist.

file - Kommando:

Der Befehl *`file`* ist ein mächtiger Befehl, um Statusabfragen an das UNIX-Filesystem zu richten. Er funktioniert genauso, wie alle **Tcl**-Befehle mit Subkommandos, bspw.:

```
file exists /test_datei.xxx

> 0
```

Hier wurde geprüft, ob *`test_datei.xxx`* im Root-Verzeinis liegt, oder nicht. Die anderen Subkommandos funktionieren analog. In den Büchern von Welch auf S.65ff. [Welch95[13]] und von Ousterhout auf Seite 115ff. [Ousterhout94 [14]] finden sich Tabellen, die alle Subkommandos enthalten.

[13] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm
[14] http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/lit_vz.htm

Kapitel 4: Tk, die Grundlagen

In *Tk* gelten alle Aussagen, die auch in **Tcl** gelten. **Tk** stellt Kommandos zur Verfügung, mit denen „GUIs“ programmiert werden können; deshalb ist es zwar ein wesentlicher Bestandteil von **Tcl**/**Tk**, aber für den *ns* Netzwerksimulator irrelevant – hier es sollen nur die wesentlichsten Elemente kurz zusammengefasst werden:

Widgets

Die deutsche Bedeutung des Wortes „widget“ wird am besten mit dem Begriff Dingens wiedergegeben. Im Bereich der „GUIs“ bezeichnet es Bestandteile der Graphikoberfläche wie z.B. Menüs, Scrollbars etc.

Es gibt folgende Widgets:

_ frame
_ label
_ button
_ checkbutton
_ radiobutton
_ menu
_ listbox
_ scrollbar
_ text
_ canvas
_ scales
_ message

Logische Struktur der Widgets untereinander

„Widgets“ sind Bestandteile eines Windows. Das oberste Window einer Anwendung, das die Widgets enthält, heißt toplevel-Window und wird durch einen Punkt „.“ symbolisiert. Das oberste Window, mit dem wir es zu tun haben, ist stets ein Frame. Er wird im einfachsten Fall folgendermaßen erzeugt:

frame .mainframe

Die Formulierung *.mainframe* nennt man auch „window path“. Soll nun innerhalb des Frames ein Button angelegt werden, so könnte man das so formulieren:

button .mainframe.button1 -text "Press me" -command "puts Hallo"

Manche „Widgets“ können selbst wieder andere „Widgets“ enthalten, wodurch sich im Prinzip eine Baumstruktur ergibt. Zu allen „Widgets“ gibt es eine mehr oder weniger große Anzahl von Optionen, mit denen sich das genaue Aussehen und Verhalten der „Widgets“ einstellen lässt. Ein Beispiel dafür ist schon in der obigen Button-Definition gegeben.

Packen der Widgets

Kommandos, die „Widgets“ definieren, bringen sie noch nicht sofort auf den Bildschirm. Dies geschieht erst durch das Kommando *pack*. Um obiges Beispiel sichtbar zu machen, ist noch folgendes Kommando nötig:

pack .mainframe .mainframe.button1

Über das *pack*-Kommando kann man die Anordnung der einzelnen „Widgets“ relativ zueinander bestimmen. Dazu hat auch *pack* eine Anzahl von Optionen.

Binding

Viele „Widgets“ sollen eine Aktion ausführen, z.B. wenn sie angeklickt werden. Ein typisches Beispiel dafür ist ein Button. Man kann einem „Widget“ ein Script zuordnen, welches ausgeführt wird, sobald ein bestimmtes Ereignis eintritt. Ereignisse (Events) kommen typischerweise von der Maus oder der Tastatur und können ebenso wie eine Aktion spezifiziert werden. Es wird also ein Kommando an ein Ereignis gebunden. Aktionen können implizit gleich zusammen mit dem „Widget“ definiert werden.

z.B.:

button .b -text "press here" -command { destroy .b }
Sie können auch explizit mit dem Kommando bind definiert werden, z.B.:

bind .b <Button-1> { destroy .b ; break }

Kapitel 5: OTcl

OTcl ist eine am MIT entwickelte Erweiterung der **Tcl**-Scriptsprache um einfache objektorientierte Sprachelemente; **OTcl** wird zur Bedienung des **ns** Netzwerksimulators eingesetzt.

Leider ist dieser Teil der Dokumentation noch nicht fertig.
Es kann daher lediglich auf externe Unterlagen verwiesen werden:

Quelle der Originaldistribution: ftp://ftp.tns.lcs.mit.edu/pub/otcl/README.html

Tutorial: ftp://ftp.tns.lcs.mit.edu/pub/otcl/doc/tutorial.html

Es wird dringend empfohlen, zumindest das (kurze und leicht verständliche!) Tutorial durchzuarbeiten.

Für weitere Informationen zu **OTcl** (Objektorientiert) verweisen wir auf:
http://nestroy.wi-inf.uni-essen.de/Lv/seminare/ws9697/zdun/inhalt.htm

Kapitel 6: ns im Detail

In diesem Kapitel werden einzelne Elemente und Features von **ns** kurz vogestellt; für entsprechende Beispiele sei auf das Tutorial in Anhang A verwiesen.

OTcl Grundgerüst

ns ist ein objektorientierter Simulator, der in C++ geschrieben wurde. Zur Bedienung wird ein **OTcl** Interpreter verwendet, welcher über die Konsole zu steuern ist. Der Simulator hat eine Klassenhierarchie in C++ sowie eine ähnliche Klassenhierarchie für den **OTcl** Interpreter. Diese zwei Hierarchien sind sich sehr ähnlich. Von der Benutzersicht aus gesehen gibt es eine eins zu eins Beziehung zwischen den Objekten beider Hierarchien. Die Wurzel der Hierarchie ist die Hierarchie von **OTcl**. Anwender erstellen neue Simulator Objekte durch den Interpreter. Diese Objekte sind innerhalb des Interpreters instantiiert und werden durch ein entsprechendes Objekt in der kompilierten Hierarchie wiedergegeben. Es gibt aber auch andere Hierarchien im C++ Code und in den **OTcl** Indizes, welche nicht in dieser Art und Weise von **OTcl** gespiegelt werden.

Konzept und Übersicht

Warum zwei Sprachen? **ns** verwendet zwei Sprachen, weil der Simulator zwei verschiedene Arten von Anforderungen erfüllen muss. Einerseits erfordert eine ausführliche Simulation von Protokollen eine Implementierungssprache, die Bytes, Paketheader und Protokollalgorithmen manipulieren kann. Für diese Aufgaben ist Laufzeitgeschwindigkeit wichtig und erforderlich. Abfertigungszeit (Durchlaufsimulationen, finden von Bugs, recompile usw.) ist in diesem Fall weniger wichtig.

Andererseits bezieht sich ein großer Teil der Netzwerkforschung auf das Ändern von Parametern oder die Änderung von Konfigurationen anhand einer bestimmten Anzahl an scripts. In diesem Fall ist Wiederholungszeit (ändern des Modells und die Wiederholung) sehr wichtig. Da die Konfiguration einmal festgelegt wurde (am Anfang) und bereits läuft, ist die Laufzeit für diesen Teil der Aufgabe weniger wichtig.

ns erfüllt beide dieser Bedürfnisse mit zwei Sprachen: C++ und **OTcl**. C++ läuft sehr schnell und ist für ausführliche Protokollimplementierungen gut geeignet, ist aber sehr schwerfällig bei Änderungen. Der Vorteil von **OTcl** hingegen ist es, dass man sehr schnell wechselwirkende Änderungen vor nehmen kann. **OTcl** ist daher ideal zur Beschreibung von Simulationskonfigurationen.

OTcl wird verwendet für: Konfigurationsänderungen, Setups, „on-time“ Einstellungen. C++ wird verwendet: für Code, der bei der Verarbeitung jedes Paketes eines Datenflusses erforderlich ist. z.B. sind Verbindungen **OTcl** Objekte, die Verzögerungen realisieren und vielleicht Verlustmodelle beinhalten. Wenn man stattdessen selbst z.B. ein neues Verlustmodell für einen Link entwickeln möchte, wird man C++ brauchen.

Anmerkung: Auf die rein für die Entwicklung mit C++ relevanten Dinge sowie die Klassenhierarchie in C++ wird in der weiteren Dokumentation nicht eingegangen, da dies für die reine Benutzung (nicht Veränderung) von **ns** nicht benötigt wird.

Die Klasse OTcl

Die Klasse **OTcl** ist die Basis Kategorie für die meisten anderen Kategorien in beiden Hierarchien. Jedes Objekt in der Kategorie **OTcl** wird vom Benutzer innerhalb des Interpreters erzeugt. Ein gleichwertiges Objekt wird dann in der kompilierten Hierarchie generiert. Diese zwei Objekte sind nah miteinander verbunden. Die Kategorie TClClass enthält alle dazu benötigen Einheiten.

Bsp.: Konfiguration eines **Tcl**Objekts:

```
set srm [new Agent/SRM/Adaptive]
$srm set packetSize_ 1024
$srm traffic-source $s0
```

Durch die Zusammenführung im **ns** ist die Kategorie Agent/SRM/Adaptive eine Unterklasse von Agent/SRM, und diese ist eine Unterklasse von SRM, welche eine Unterklasse von **Tcl**Object ist. Die dementsprechend kompilierte Hierarchie ist abgeleitet von SRMAgent, und diese Klasse ist abgeleitet von SRM, und diese von **Tcl**Object.

Erstellen und Löschen von TclObjekten

Wenn der User ein neues Objekt erstellen oder löschen möchte, dann muss er die Prozeduren new{} bzw. delete{} verwenden. Diese Prozeduren sind in ~tcltcl/tcl-objects definiert. Sie können dazu verwendet werden, um in allen Klassen Objekte zu erstellen bzw. zu löschen. In diesem Abschnitt werden die intern durchgeführten Tätigkeiten beschrieben, wenn ein Objekt generiert wird.

Indem man new{} verwendet, wird vom Interpreter ein gespiegeltes Objekt erzeugt. init{} stellt das Objekt bereit. **ns** ist für das Kompilieren alleine verantwortlich. Der Interpreter gibt eine Art Referenz auf das Objekt zurück, damit mit dem Objekt weitergearbeitet werden kann.

Bidirektionale Variablen

In den meisten Fällen wird der Zugang zu kompilierten Variablen verwährt, und der Zugang zu Objektattributen ist meistens eingeschränkt. Es gibt allerdings die Möglichkeit, bidirektionale Variablen zu generieren, sodass die gespiegelten Variablen auf die Daten der kompilierten Variablen zugreifen können. Die Bidirektionalität wird vom Compiler hergestellt, sofern die Variablen instantiiert wurden. **ns** unterstützt 5 Variablentypen, mit denen das möglich ist: reals, bandwidth valued variables, time valued variables, integers, und booleans.

z.B. sind alle folgenden Beispiele gleichwertig:

- Real time und Integer Variablen:

  $object set realvar 1.2e3
  $object set intvar 12

- Bandbreite wird als realer Wert spezifiziert, beliebig angefügt durch ein k oder K (Kilo-Einheiten) oder m oder M (Mega-Einheiten). Ein abschließendes wahlweise freigestelltes Suffix von B zeigt an, dass die ausgedrückte Einheits in Bytes pro Sekunde ist. Der Rückgabewert ist die Bandbreite, die in Bits pro Sekunde ausgedrückt wird.

  $object set bwvar 1.5m
  $object set bwvar 1.5mb
  $object set bwvar 1500k
  $object set bwvar 1500kb
  $object set bwvar .1875MB
  $object set bwvar 187.5kB
  $object set bwvar 1.5e6

- Die Zeit wird als realer Wert verwendet, angefügt wird ein m für Millisekunden, ein n für Nanosekunden und p für Picosekunden.

  $object set timevar 1500m
  $object set timevar 1.5
  $object set timevar 1.5e9n
  $object set timevar 1500e9p

- Boolesche Werte können entweder als Integer Werte ausgedrückt werden (1 = true, 0 = false, wie in C bzw. C++) oder durch „true“ und „false“ oder T oder t bzw. F oder f.

  $object set boolvar t
  $object set boolvar true
  $object set boolvar 1
  $object set boolvar false
  $object set boolvar junk
  $object set boolvar 0

So setzt man also z.B. Attribute eines Agenten:

Agent/SRM/Adaptive set pdistance_ 15.0
Agent/SRM set pdistance_ 10.0
Agent/SRM set lastSent_ 8.345m
Agent set ctrlLimit_ 1.44M
Agent/SRM/Adaptive set running_ f

Überblick: Allgemeine ns Befehle

Es folgen einige allgemeine **ns** Befehle, die in erster Linie mit dem Scheduler (wann tritt welches Ereignis ein?) und Datenformat zu tun haben. Besonders wichtige Befehle sind fett dargestellt.

Kommandozeilenargumente, um den **ns** zu starten:
`ns <otclfile> <arg> <arg>..`

Die folgende Liste enthält Kommandos, welche bei der Simulation verwendet werden. Der Simulator (**ns**) wird über den **ns** Interpreter, einer Erweiterung eines Teils der otclsh aufgerufen. Eine Simulation wird durch einen **OTcl** Index definiert. Einige Beispiele der **OTcl** Indizes können im ns/tcl/ex Verzeichnis gefunden werden.

scripts:
`set ns_ [new Simulator]`
Dieses Kommando kreiert eine neue Instanz („**ns_**“) eines Simulatorobjektes.

`set now [$ns_ now]`
Der Scheduler verfolgt die Zeit in einer Simulation.

`$ns_ halt`
Dieser Befehl stoppt den Scheduler bzw. versetzt ihn in einen Pausen-Modus.

`$ns_ run`
Startet den Scheduler.

`$ns_ at <time> <event>`
Das plant ein Ereignis <event> (welches normalerweise ein Stück des Codes ist) welches zum in <time> angegebenen Simulationszeitpunkt ausgeführt wird.

Beispiel:
```
$ns_ at $opt(stop) "puts ¨ ns EXITING..¨ ; $ns_ halt"
```
oder
```
$ns_ at 10.0 "$ftp start"
```

`$ns_ cancel <event>`
Unterbricht ein Ereignis.

`$ns_ create-trace <type> <file> <src> <dst> <optional arg: op>`
Das kreiert ein trace-object von type <type> zwischen <src> und <dst> Objekten und fügt dies zu einem trace-object <file> hinzu um die Ausgabe schreiben zu können. Wenn op als "**nam**" definiert ist, wird ein **nam** tracefiles generiert. Andernfalls wird ein **ns** tracefiles erzeugt. Mehr zu trace files folgt in einem eigenen Abschnitt gegen Ende dieses Kapitels.

`$ns_ flush-trace`
Flushed den gesamten trace-file Schreibpuffer.

`$ns_ gen-map`
Das speichert die gesamte Information (wie nodes, node components, links usw.) welche für eine gegebene Simulation angelegt wurde.

`$ns_ use-scheduler <type>`

Wird verwendet um den type des Schedulers auszuwählen. Die verschiedenen Typen eines Schedulers sind folgende: List, Calendar, Heap and RealTime. Zur Zeit wird Calendar als default Einstellung verwendet.

.

```
$ns_ after <delay> <event>
```

Scheduled ein <event> um nach einer gewissen Zeit (<delay>) ausgeführt zu werden.

.

```
$ns_ clearMemTrace
```

Wird für Speicher debugging verwendet.

```
$ns_ is-started
```

Der Wert ist true wenn der Simulator bereits läuft, ansonsten false.

```
$ns_ create_packetformat
```

Das setzt das Packetformat des Simulators.

Netzwerkknoten (nodes)

Dieser Abschnitt beschreibt, wie man einfache Topologien im **ns** erstellen kann. Beginnen werden wir mit den Grundlagen zu Netzwerkknoten.

Erinnern wir uns daran, dass man eine Instanz der Klasse Simulator benötigt, um die Simulation zu kontrollieren bzw. damit sie überhaupt funktionieren kann. Diese Klasse bietet Prozeduren, um eine Topologie zu erstellen und kontrollieren. Intern werden alle Referenzen von allen Elementen gespeichert. Wir beginnen bei unserer Beschreibung der Prozeduren mit der Klasse Simulator, danach folgen die Instanzen der Prozedur node (Knoten) in der Klasse um den Zugriff auf verschiedene Knoten zu gewährleisten.

Die Prozeduren und Funktionen, die in diesem Abschnitt verwendet werden können unter ~*n*s/tcl/lib/ns-lib.tcl, ~*n*s/tcl/lib/ns-node.tcl, ~*n*s/tcl/lib/ns-rtmodule.tcl, ~*n*s/rtmodule.{cc,h}, ~*n*s/classifier.{cc, h}, ~*n*s/classifier-addr.cc, ~*n*s/classifier-mcast.cc, ~*n*s/classifier-mpath.cc, und ~*n*s/replicator.cc gefunden werden.

Erstellen eines Knotens

```
set ns [new Simulator]
$ns node [<hier_addr>]
```

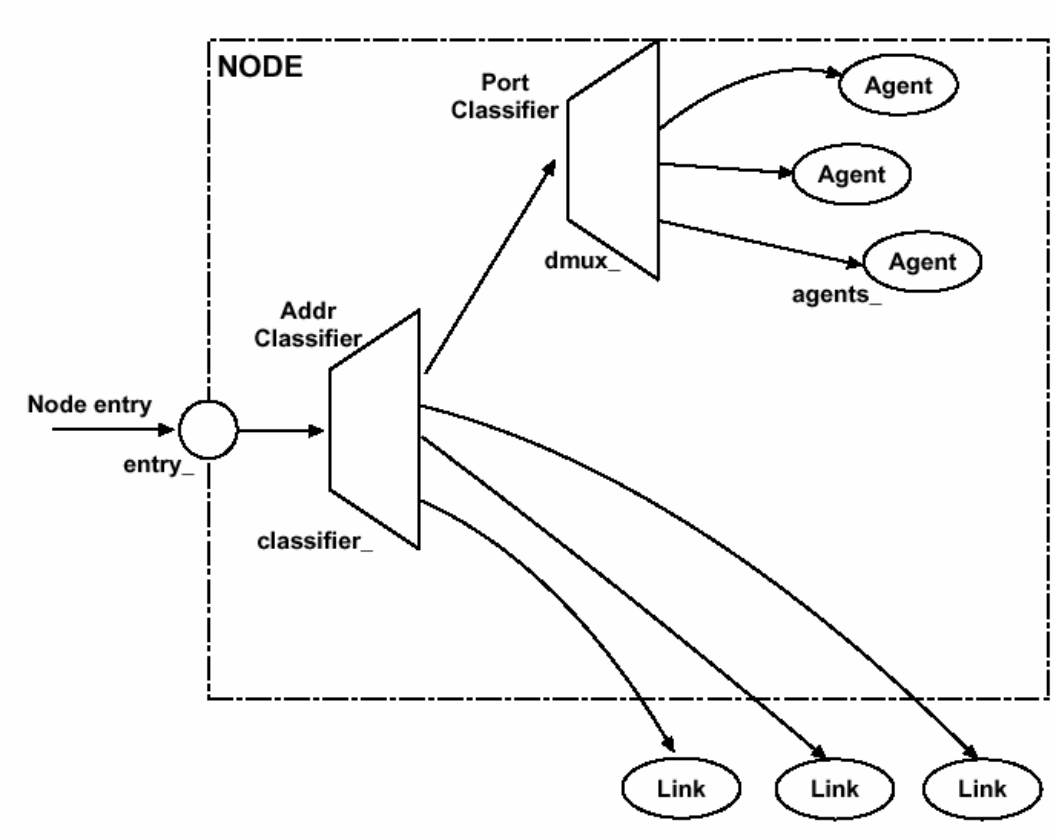
Diese Methode der Klasse node konstruiert einen Knoten. Der Knoten (node) ist eine allein stehende Klasse in **OTcl**. Die meisten Komponenten des Knotens (node) sind **Tcl**Objekte. Die typische Struktur eines (unicast) Knotens wurde oben mit den 2 Codezeilen gezeigt. Die einfache Struktur besteht aus zwei **Tcl**Objekten: einer Adressklassifizierung (classifier_) und einer Portklassifizierung (dmux_). Die Funktionen dieser zwei Klassifikationen sind es, die eingehenden Daten (Pakete) zu den einzelnen Agents oder Links korrekt und richtig zu dirigieren.

Alle Knoten beinhalten also folgende Komponenten:

- eine Adresse (id_)

- Eine Liste seiner Nachbarn (neighbor_)
- eine Liste der Agenten (agent_)
- Einen Knotentypidentifizierer (nodetype_)
- Ein Routingmodul (wird weiter unten beschrieben)

Wenn <hier_addr> gegeben ist, dann wird diese Adresse dem Knoten hinzugefügt. <hier_addr> muss nur verwendet werden, wenn die hierarchische Adressierung eingeschaltet ist (mit `set-address-format hierarchical{}` oder `node-config –addressType hierarchical{}`).

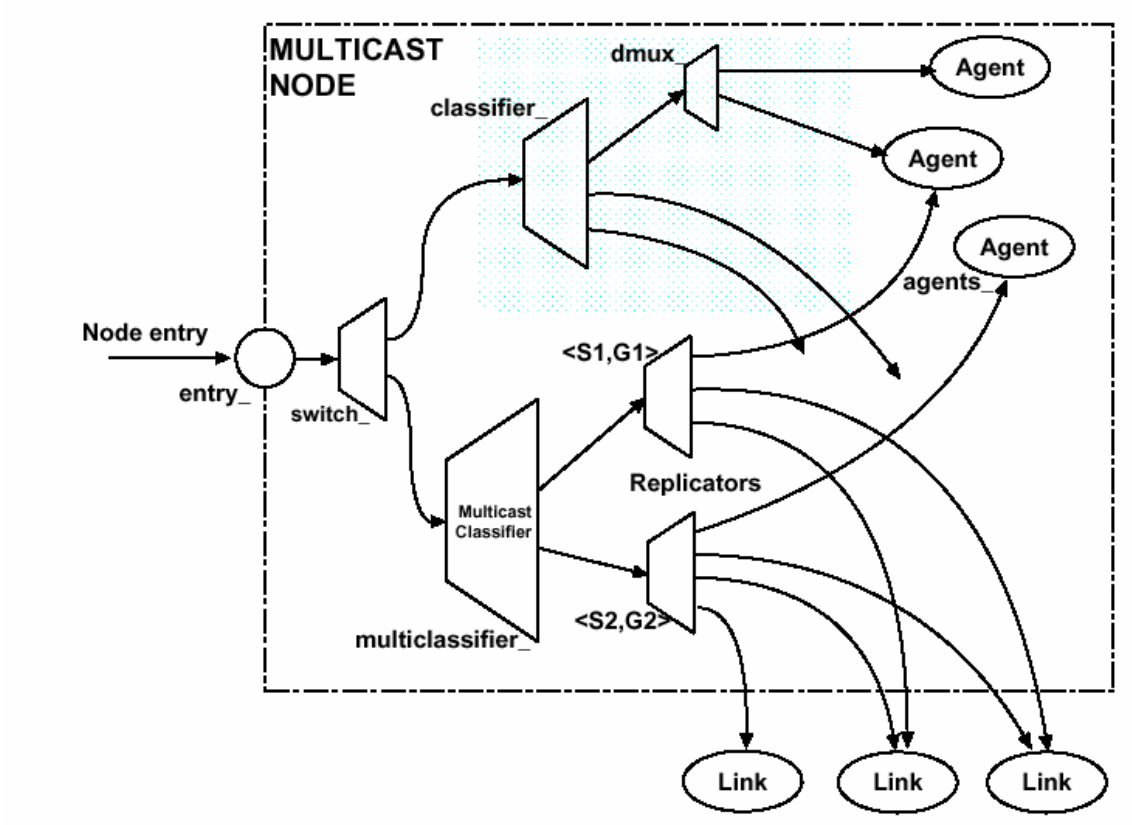


Struktur eines (unicast) Knotens

Anm.: entry_ und classifier_ sind einfache Variablen und keine Objekte.

Die default Einstellung des Simulators ist unicast. Wenn man multicast verwenden möchte, dann sollte man die Option –multicast on beim Erzeugen des Simulator Objekts verwenden:

```
set ns [new Simulator –multicast on]
```



Interne Struktur eines multicast Knotens

Konfigurieren eines Knotens

Folgende Typen von Prozeduren werden verwendet, um einen Knoten zu konfigurieren:

- Kontrollfunktion
- Adressen und Port Management, unicast routing Funktion
- Agent Management
- Hinzufügen von Nachbarn

Kontrollfunktionen

`$node entry`
gibt den Eintrittspunkt eines Knotens zurück. Das ist das 1. Element, welches die Pakete übernimmt und auf diese zugreift. Das Knoten-Attribut entry_ speichert die Referenz des Elements. Bei unicast Knoten ist dies der address classifier welcher auf die höheren Bits der Zieladresse schaut.

`$node reset`
setzt alle Agenten auf dem Knoten in den Ausgangszustand zurück.

Adressen- und Portnummern-Management

`$node id`
gibt die Knotennummer zurück. Diese Nummer wird automatisch angelegt und hoch gezählt. Jeder Knoten, der neu erzeugt wird, bekommt eine neue Nummer zu gewiesen – also zum Beispiel: node1 – id 1, node 2 – id 2 usw. Die Klasse Simulator speichert intern ein Array Node_. Die Indizes in diesem Array entsprechen den node id´s und beinhalten eine Referenz zum jeweiligen Knoten.

`$node agent <port>`
gibt den Handler des Agent zurück, welcher gerade den spezifizierten Port verwendet. Wird kein Agent gefunden, erhält man einen Nullstring als Rückgabewert.

`$node alloc-port`
gibt den nächsten möglichen freien Port zurück. Sie verwendet ein Attribut np_, um den nächsten freien Platz zu finden.

`$node add-route <destination id> <TclObjekt>`
wir bei unicast und multicast routing verwendet. **Tcl**Objekt ist der Eintrag von dmux_ – dem Portdemultiplexer des Knotens. Wenn die Ziel ID (Destination) gleich der Knoten ID ist, dann ist das meistens der Kopf (header) eines Links, welcher die Daten an die Zieladresse schickt. Allerdings könnten es genauso Einträge andere Klassifizierer sein.

`$node add-routes <Destination id> <TclObjects>`
wird verwendet um mehrere Routen auf das gleiche Ziel zu legen. Diese werden simultan benutzt, und zwar mittels des von Betriebssystemen her bekannten „round robin“ Verfahrens; damit wird die Bandbreite vergrößert. Das hat aber nur Gültigkeit, wenn das Attribut multiPath auf 1 gesetzt wurden.

Das Pendant zur internen Prozedur add-routes{} ist delete-routes{}. Diese Prozedur nimmt die ID, die Liste der **Tcl**Objekte und die Referenz zum Simulator Null Agent. Es werden alle **Tcl**Objekte gelöscht, welche in der Liste stehen. Detailiertes dynamisches routing verwendet zwei Methoden:

1. Die interne Prozedur: init-routing{} setzt das Attribut multiPath_. Außerdem wird eine Referenz zum Routenhandler Objekt dieses Knotens im Attribut rtObject_ angelegt. Die Prozedur rtObject{} gibt den Zugriff auf das Routing-Objekt des Knotens zurück.
2. Die Prozedur intf-changed{} wird vom „network dynamic code“ aufgerufen, wenn sich der Status eines Links ändert.

Weitere Befehle

`$ns_ node-config -<config-parameter> <optional-val>`
Dieses Kommando wird verwendet, wenn man den Knoten konfigurieren möchte. Die unterschiedlichen Konfigurationsparameter sind: adressingType (art der Adresse), verschiedene Arten vom Typ des Netzwerkes: stack components – mit dessen Hilfe das Tracing ein bzw. ausgeschaltet wird, mobileIP (ein – bzw. ausschalten), usw.

`$node node-addr`
Gibt die Adresse des Knotens zurück.

`$node attach <agent> <optional:port_num>`
Fügt den Agenten zu diesem Knoten hinzu. Alternativ kann eine Portnummer angegeben werden, welche dann dem Agenten hinzugefügt wird.

`$node detach <agent> <null_agent>`
Das Pendant zu attach. Der Agent wird gelöscht und anstelle des gelöschten Agenten wird ein Null-agent installiert.

`$node neighbors`
Gibt die Liste der Nachbarn des Knotens zurück.

`$node add-neighbor <neighbor_node>`
Fügt einen Nachbarn hinzu.

`$node alloc-port <null_agent>`
Gibt den nächsten freien bzw. möglichen Port zurück.

`$node incr-rtgtable-size`
Das Attribut rtsize_ wird verwendet um die Größe der routing-Tabelle für jeden Knoten zu definieren.

Links (Verbindungen)

Im vorigen Abschnitt haben wir beschrieben, wie man Knoten erzeugen kann. Jetzt gehen wir daran, die Knoten mit Links zu verbinden und die Topologie dadurch zu vervollständigen. In diesem Abschnitt behandeln wir einfache Punkt zu Punkt (point to point) Verbindungen. **ns** unterstützt eine Vielfalt anderer Medien – der Simulator kann beispielsweise multi-access LANs oder auch Satellitenverbindungen simulieren. Derartige Medien werden in diesem Dokument allerdings nicht behandelt.

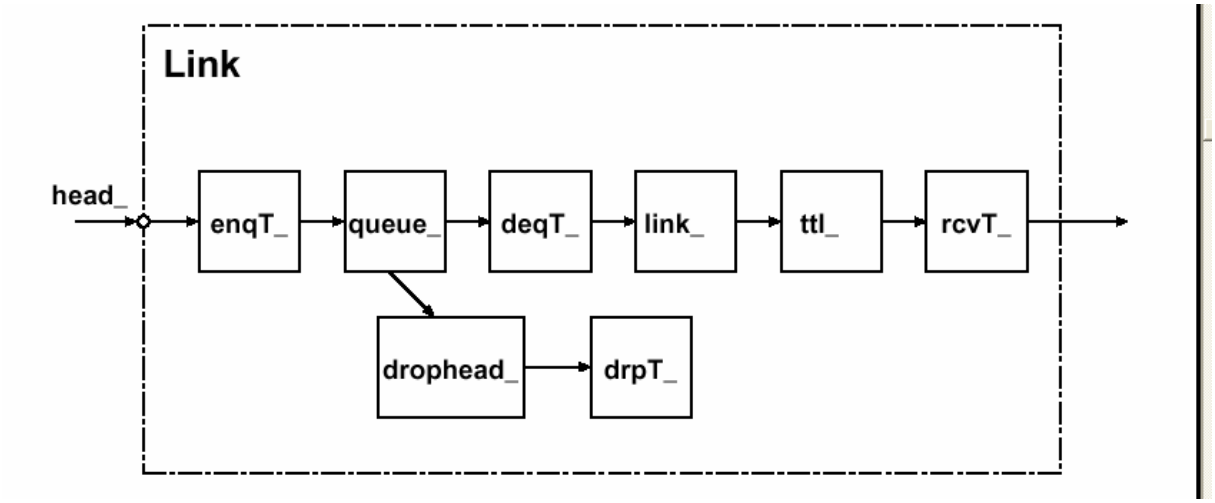
Wir beginnen mit der Beschreibung der Kommandos, die nötig sind, um einen Link zu erzeugen. So wie ein Knoten aus einer bestimmten Anzahl von Klassifizierern besteht, besteht ein Link aus einer Reihe von Verbindungssequenzen, welche später etwas genauer beschrieben werden. Die Klasse Link ist eine für sich allein stehende Klasse in **OTcl**. Die Klasse SimpleLink bietet die Möglichkeit an, zwei Knoten mit einem Punkt zu Punkt Link zu verbinden. Im **ns** wird die Prozedur simplex-link verwendet, um eine unidirektionalen Link zwischen zwei Knoten zu erzeugen. Hier die Syntax:

```
set ns [new Simulator]
$ns simplex-link <node0> <node1> <bandwidth> <delay> <queue_type>
```

Dieses Kommando erzeugt einen Link vom Knoten <node 0> zum Knoten <node 1> mit den Attributen Bandbreite <bandwidth> und Verzögerung <delay>. Der Link verwendet eine Queue mit dem Typ <queue_ type> (z.B. „DropTail“ für eine gewöhnliche FIFO Queue). Die Prozedur fügt zusätzlich noch einen TTL Checker (Überprüfer) hinzu (TTL = „Time To Live“, entspricht dem gleichnamigen Feld im IP header).

Ein Link beinhaltet 5 Attribute:

- **head_ :** Eintrittspunkt des Links, er zeigt auf das erste Objekt im Link.
- **queue_ :** Referenz auf das Haupt-Queue-Element des Links. Einfache Links haben nur eine Queue pro Link. Andere bzw. komplexere Typen haben meist mehrere Queues Elemente.
- **link_ :** Referenz auf ein Element, das den eigentlichen Link modelliert. Das Verhalten dieses Elements hängt von der Bandbreite und Verzögerung des Links ab.
- **ttl_ :** Referenz auf ein Element, welches die TTL in jedem Paket manipuliert.
- **drophead_ :** Referenz auf ein Objekt, welches der Kopf des Queue Elements ist (über das Pakete verworfen werden).



Aufbau eines Links

Die Attribute, die mit „T_“ enden, speichern sog. „Trace“-Elemente; Details zu „Tracing“ folgen am Ende dieses Kapitels.

Methoden der Klasse Link

`$link head`
Gibt das Attribut head_ eines Links zurück.

`$link add-to-head <connector>`
Das erlaubt dem <connector> Objekt, dass es nun über den head_ angesprochen werden kann.

`$link link`
Gibt das Attribut link_ zurück.

`$link queue`
Gibt das Attribut queue_ zurück.

`$link cost <c>`
Setzt Link Kosten (für Routing) von <c>.

`$link cost?`
Gibt die cost Variable eines Links zurück.

`$link if-label?`

Gibt das Netzwerkinterface zurück.

`$link up`
Setzt den Link Status auf "up".

`$link down`
Setzt den Link Status auf "down".

`$link up?`
Gibt den Link Status zurück.

`$link all-connectors op`
Dieses Kommando erlaubt spezielle Operationen auf allen „Connectors" (Elemente, die Pakete innerhalb eines Links weiterleiten und Operationen darauf ausführen) eines Links.

`$link install-error <errmodel>`
Diese Methode installiert ein Fehlermodul (zur Erzeugung von Verbindungsfehlern) nach einem Link Element.

Methoden der Klasse SimpleLink

Bei SimpleLink handelt es sich um eine abgeleitete Klasse der Link-Basisklasse; sie wird verwendet, um einen einfachen unidirektionalen Link zu repräsentieren. In dieser Klasse gibt es keine statischen Variablen bzw. Konfigurationsmöglichkeiten (Parameter).

`$simplelink enable-mcast <src> <dst>`
Dieser Befehl schaltet multicast für den Link ein.

`$simplelink trace <ns> <file> <optional:op>`
Erzeugt trace-Objekte für den Link. ns ist eine Referenz auf das verwendete Netzwerksimulator-Objekt. Wenn op als „**nam**" spezifiziert ist, werden **nam** trace-Files generiert.

`$simplelink nam-trace <ns> <file>`
Setzt **nam**-Tracing im Link.

`$simplelink trace-dynamics <ns> <file> <optional:op>`
Setzt das Tracing speziell für dynamische Links.

`$simplelink init-monitor <ns> <qtrace> <sampleInterval>`
Fügt einen „Monitor" ein (ein Objekt, welches es erlaubt, die Queuelänge zu überwachen).

`$simplelink attach-monitors <insnoop> <outsnoop> <dropsnoop> <qmon>`
Initialisiert einen „Monitor".

`$simplelink dynamic`
Setzt das dynamic Flag für den Link.

`$simplelink errormodule <args>`
Fügt ein Fehlermodul vor die Queue ein.

`$simplelink insert-linkloss <args>`
Fügt ein Fehlermodul nach der Queue ein.

Methoden der Klasse ns

`$ns_ simplex-link <node1> <node2> <bw> <delay> <qtype> <args>`
Dieses Kommando erzeugt einen unidirektionalen Link zwischen Knoten 1 und Knoten 2.

`$ns_ duplex-link <node1> <node2> <bw> <delay> <qtype> <args>`
Dieser Befehl erzeugt einen bidirektionalen Link zwischen Knoten 1 und Knoten 2 mit Hilfe zweier Simplex Links.

`$ns_ duplex-intserv-link <n1> <n2> <bw> <dly> <sched> <signal> <adc> <args>`
Erzeugt einen duplex-Link zwischen n1 und n2 mit dem Queue Typ von „IntServ“.

`$ns_ link <node1> <node2>`
Dieser Befehl liefert das Link-Objekt zwischen den Knoten 1 und 2 zurück.

`$ns_ simplex-link-op <n1> <n2> <op> <args>`
Wird verwendet, um Attribute eines simplex Links zu setzen. Attribute können in diesem Fall: Farbe, Beschriftung, Queue Position usw. sein.

`$ns_ duplex-link-op <n1> <n2> <op> <args>`
Wird verwendet, um Attribute eines duplex Links zu setzen.

`$ns_ link-lossmodel <lossobj> <from> <to>`
Diese Funktion erzeugt Verluste.

`$ns_ lossmodel <lossobj> <from> <to>`
Wird verwendet, um ein Verlust-Modell in herkömmliche Links einzubauen.

`$ns_ register-nam-linkconfig <link>`
Das ist eine interne Prozedur, welche von „$link orient“ verwendet wird, um zu registrieren in welchem Link das erzeugt werden soll.

`$ns_ remove-nam-linkconfig <id1> <id2>`
Diese Prozedur wird verwendet, um Duplikationen von Links zu löschen.

Queues

Das Verhalten eines Links wird weitgehend durch das Verhalten seiner Queue(s) bestimmt. Eine Queue dient als Puffer für Pakete, die sich durch die simulierte Topologie bewegen; möglicherweise werden diese Pakete verworfen oder markiert. Die Queue-Basisklasse hat folgende drei Attribute:

- **limit_** : Die Queuelänge in Paketen.
- **blocked_** : dieses Attribut ist true falls die Queue blockiert ist (d..h., es ist nicht möglich, ein Paket zum nächsten angeforderten Knoten weiterzuschicken). Standardeinstellung: false
- **unblock_on_resume_** : zeigt an, ob eine Queue nicht mehr blockiert sein soll wenn das letzte Paket übermittelt (aber nicht zwingend empfangen) wurde. Standardeinstellung: true.

Abgeleitete Klassen von Queue sind:

- **DropTail:** eine einfache FIFO Queue ohne Methoden, Konfigurationsparameter oder Zustandsvariablen.
- **FQ:** implementiert Fair Queuing. Es gibt keine spezifischen Methoden und Zustandsvariablen für diese Klasse - lediglich einen Konfigurationsparameter namens „secsPerByte_“.
- **SFQ** (Stochastic Fair Queuing), **DRR** (Deficit Round Robin), **RED** (Random Early Detection), und **CBQ** (Class-Based Queuing) sind weitere unterstützte Queuing Verfahren. Details zur Funktionsweise dieser Verfahren[15] würden den Rahmen dieser Dokumentation sprengen; für Konfigurationsparameter, Zustandsvariablen und Methoden sei auf das ns manual verwiesen.

Folgende ns-Methoden beziehen sich auf die Handhabung von Queues:

`$ns_ queue-limit <n1> <n2> <limit>`
Dieser Befehl beschränkt die maximale Puffergröße der Queue zwischen den Knoten <n1> und <n2>.

`$ns_ trace-queue <n1> <n2> <optional:file>`
Dieser Befehl installiert trace Objekte um Ereignisse in der Queue zu protokollieren. Wenn kein file übergeben wird, wird traceAllFile_ verwendet, um die Ereignisse zu speichern.

`$ns_ namtrace-queue <n1> <n2> <optional:file>`
Wie trace-queue, aber mit tracing im nam-Format.

`$ns_ monitor-queue <n1> <n2> <optional:qtrace> <optional:sampleinterval>`
Dieser Befehl fügt Objekte ein, die es erlauben, die Queuelänge zu überwachen. Es wird ein Verweis auf das Objekt zurückgegeben, das abgefragt werden kann um die durchschnittliche Queuelänge zu ermitteln. Der Standardwert für sampleinterval ist 0.1.

Agenten und Anwendungen

Agenten repräsentieren Endobjekte, mit denen Netzwerkprotokolle realisiert werden. Agenten werden in Implementierungen von verschiedenen Protokoll Layern verwendet. Es werden sehr viele verschiedene Arten von Agenten vom Simulator unterstützt; darunter sind u.a. diverse TCP-Implementierungen mit unterschiedlichen Empfängern, ein UDP-Agent und das „Real Time Protocol“ (RTP).

Die Basisklasse „Agent“ hat eine Reihe von Attributen, die sich grundsätzlich auf den Datenfluss zwischen einem Sender und einem Empfänger beziehen: das Attribut fid_ etwa kann zur Identifikation des Datenflusses dienen und findet sich in trace-files wieder (siehe Abschnitt über Tracing). Diese Attribute eines Agents können vom Agent oder auch vom Benutzer modifiziert werden; es kann auch sein, dass nicht alle Variablen verwendet werden.

[15] Empfohlene Referenz: Grenville Armitage, „Quality of Service in IP Networks“, Macmillan Technical Publishing, USA 2000.

- **fid_** Fluß ID
- **prio_** Priorität (vgl. Priority-Feld im IP-Header)
- **agent_addr_** Adresse dieses Agenten
- **agent_port_** Port dieses Agenten
- **dst_addr_** Ziel Adresse dieses Agenten
- **dst_port_** Ziel Port dieses Agenten
- **flags_** Flags zum Datenfluß (z.B. Explicit Congestion Notification (ECN))
- **ttl_** vgl. „Time To Live“ (TTL) im IP-Header; Standardwert: 32.

Grundsätzlich wird man in Simulationen meistens nur einen oder höchstens zwei Agenten an einem Knoten installieren; sollten es mehrere sein, werden es kaum gleichartige Agenten sein. Darum muß man sich im Normalfall nicht selbst um Ports kümmern. Diese Modellierung ist allerdings im **ns** vorhanden und kann über die Methoden „port“ und „dst_port“ angesprochen werden. Die wichtigste Methode der Agent Klasse ist jedenfalls:

`attach-source <stype>`
zur Verbindung einer Anwendung mit dem Agenten.

In folgendem Beispiel wird ein einfacher TCP Agent inklusive entsprechendem Empfänger auf den Knoten n0 und n1 erzeugt und mit einer ftp-Anwendung verbunden:

```
set tcp [new Agent/TCP] ;# erzeugt einen Sender Agenten
$tcp set fid_ 0 ;# setzt die flow id - scheint später im Trace file auf
set sink [new Agent/TCPSink] ;# erzeugt einen Empfangs Agenten
$ns attach-agent $n0 $tcp ;# fügt den Sender an n0
$ns attach-agent $n1 $sink ;# fügt den Empfänger an n1.
$ns connect $tcp $sink ;# eröffnet die TCP Verbindung.
set ftp [new Application/FTP] ;# erstellt eine FTP Quelle
$ftp attach-agent $tcp ;# verbindet FTP mit dem TCP Sender
$ns at 1.2 "$ftp start" ;# teilt FTP mit, dass es nach 1.2 Sekunden startet
```

TCP & Co.

Es gibt folgende TCP-Implementierungen:

- **TCP** – ein “Tahoe” TCP Sender
- **TCP/Reno** – ein “Reno” TCP Sender
- **TCP/Newreno** – ein modifizierter “Reno” Sender (ändert das retransmission-Verhalten)
- **TCP/Sack1** – ein „Selective Acknowledgments“ (SACK) TCP Sender
- **TCP/Fack** – ein „Forward Acknowledgments“ (FACK) TCP Sender
- **TCP/FullTcp** – eine voll funktionstaugliche bidirektionale TCP Implementierung
- **TCP/Vegas** – ein “Vegas” TCP Sender
- **TCP/Vegas/RBP** – ein “Vegas” TCP Sender mit “in Raten basierendem Vorwärtsschreiten” **TCP/Asym** – ein experimenteller “Tahoe” TCP Sender für asymmetrische Links
- **TCP/Reno/Asym** - ein experimenteller “Reno” TCP Sender für asymmetrische Links
- **TCP/Newreno/Asym** - ein “NewReno” TCP Sender für asymmetrische Links
- ein TCP Empfänger mit verzögerten Acknowledgments
- **TCPSink/Asym** – ein TCP Empfänger für asymmetrische Links

- **SimpleTcp** – ein stark vereinfachter TCP-Sender (u.a. ohne retransmission von verlorerene Paketen!), der als „Unterlage“ für TCPApp verwendet werden kann; weitere Details folgen im Abschnitt zum Senden von Anwendungsdaten

All diese Implementierungen (bis auf eine Ausnahme, auf die wir später zu sprechen kommen) sind unidirektional: kein Agent ist Sender und Empfänger zugleich. Möchte man eine bidirektionale Kommunikation, so muss man auf einem Knoten sowohl einen Sender als auch einen Empfänger installieren. Der Empfänger zu den Sendern **TCP**, **TCP/Reno** und **TCP/Newreno** ist **TCPSink**. Sollen Bestätigungen verzögert werden, kann stattdessen **TCPSink/DelAck** verwendet werden. Der Empfänger zu TCP/Sack1 heißt **TCPSink/Sack1**, die entsprechende Variante mit verzögerten Bestätigungen ist **TCPSink/Sack1/DelAck**. Für alle anderen TCP-Implementierungen und weiteren Agenten gilt: wenn nichts anderes angegeben wird, ist man gut beraten, es Empfängerseitig mit dem gleichen Agenten wie beim Sender zu versuchen.

Die meisten TCP Sender haben zumindest die folgenden wichtigsten Parameter (Attribute):

- **window_ –** die obere Limit für das TCP-Sendefensters
- **maxcwnd_ –** das obere Limit für das Staukontrollfenster des TCP-Senders
- **windowInit_ –** die Anfangsgröße des TCP-Sendefensters für Slow Start
- **windowThresh_ –** Verstärkungsfaktor für den exponentiellen Durchschnitts-Filter, der verwendet wird, um awnd zu berechnen (siehe unten). Zur Untersuchung unterschiedlicher Fenstergrößen-Anpassungsverfahren
- **overhead_ –** der Reichweite einer Zufallsvariable, welche für die Verzögerung für jedes Ausgangspaket verwendet wird, um Schwankungen am Sender nachzubilden und so Phaseneffekte zu vermeiden (kein realistisches CPU-Last-Modell)
- **ecn_ –** Explicit Congestion Nofication: wird auf true gesetzt, um explizit eine Benachrichtigung von verlorenen Paketen zu erhalten, wenn Active Queue Management verwendet wird
- **packetSize_ –** die Größe in Bytes für alle Pakete in dieser Quelle
- **tcpTick_ –** die TCP Clock wird zum Messen der Umlaufzeit (RTT) verwendet
- **maxburst_ –** wird auf Null gesetzt, wenn man diese Funktion ignorieren möchte. Andernfalls gibt dieser Wert die maximale Anzahl der Pakete an, welche die Quelle als Reaktion auf ein einfaches ACK senden kann
- **slow_start_restart_ –** wird auf 1 gesetzt, um in den Slow Start Modus zu gehen, sobald die Verbindung nicht genutzt wird (defaultmäßig aktiviert)
- **MWS –** die maximale Fenstergröße in Paketen für die TCP Verbindung

... und die folgenden wichtigsten Zustandsvariablen (lesbare Attribute):

- **dupacks_ –** Anzahl der gesehenen doppelten ACKs, bevor neue Daten bestätigt wurden
- **seqno_ –** höchste Sequenznummer der Daten von einer Daten Quelle über TCP
- **t_seqno_ –** Sequenznummer der aktuellen Übertragung
- **ack_** Höchste Bestätigungsnummer vom Empfänger
- **cwnd_** – aktueller Wert des Staufensters
- **awnd_ –** momentaner Wert einer tiefpassgefilterten Version des Staufensters (zur Untersuchung unterschiedlicher Fenstergrößen-Anpassungsverfahren)

- **ssthresh_ –** momentaner Wert des Slow-Start Schwellwerts (threshold, der den Übergang von Slow-Start auf Congestion-Avoidance bestimmt)
- **rtt_ –** geschätzte Umlaufzeit
- **srtt_ –** geschätzte und durch einen exponentiell gewichteten gleitenden Durchschnitt geglättete Durchlaufzeit
- **rttvar_** – geschätzte Standardabweichung der Umlaufzeit (RTT)
- **backoff_** exponentiale backoff Konstante der Umlaufzeit (RTT)

TCP Sinks (Empfänger) haben keine Methoden oder Zustandsvariablen und lediglich einen Konfigurationsparameter:

packetSize_ – die Größe in Bytes, die für alle ACKs verwendet werden soll.

In der TCPSink/Sack1 Subklasse ist zusätzlich der Parameter **maxSackBlocks_** spezifiziert, der die maximale Anzahl an Datenblöcken angibt, die in einer SACK Option bestätigt werden können.

Weitere Agenten:

- **UDP** – ein einfacher UDP Agent (benötigt wie auch im „richtigen Leben“ keinen besonderen Empfänger)
- **RTP** – ein RTP Sender und Empfänger
- **RTCP** – ein RTCP Sender und Empfänger
- **LossMonitor** – eine Datensenke welche Informationen zu erhaltenen Pakete mitprotokolliert und daher zur Ermittlung von Verlust verwendet werden kann (für mehr Details siehe Abschnitt über Tracing bzw. Appendix A). Ein LossMonitor kann z.B. mit einem UDP-Sender verbunden werden. Die Methode “clear” setzt die Sequenznummer auf -1. Die lesbaren Attribute sind:
  - **nlost_** – Anzahl der Pakete welche verloren gegangen sind
  - **npkts_** – Anzahl der Pakete welche empfangen wurden
  - **bytes_** – Anzahl der Bytes die empfangen wurden
  - **lastPktTime_** – Zeit, zu der das letzte Paket empfangen wurde
  - **expected_** – die Sequenznummer des nächsten erwarteten Paketes
- **Null** – eine Datensenke entsprechend dem von Unix-Systemen bekannten /dev/null; es gibt keine Methoden, die für diese Klasse spezifizuert sind. Attribute sind _ sport_ und _ dport_
- **rtProto/DV** – Distanz-Vektor Routing Protokoll Agent.

Anwendungen

Alle Anwendungsobjekte werden über die Methoden „start“ und „stop“ gesteuert und mit der Methode „attach“ mit einem darunter liegenden Agenten verbunden (siehe vorhergehendes Beispiel zu FTP über TCP). Es gibt zwei Arten von Anwendungsobjekten in **ns**: Verkehrsgeneratoren und simulierte Anwendungen. Es gibt vier Arten von Verkehrsgeneratoren: Exponential, Pareto, CBR and Traffic Trace.

Application/Traffic/Exponential
Exponential Objekte erzeugen Ein/Aus Verkehr. Während "ein" Perioden werden Pakete mit einer konstanten Burst-Rate erzeugt. Während "aus" Perioden wird kein Verkehr erzeugt. “Ein” und “aus” Zeiten werden einer Exponentialverteilung entnommen. Konfigurationsparameter sind:

- **PacketSize_** – konstante Größe erzeugter Pakete
- **burst_time_** – durchschnittliche “ein” Zeit
- **idle_time_** – durchschnittliche “aus” Zeit
- **rate_** – Senderate während der “ein” Zeit

Application/Traffic/Pareto
Diese Objekte erzeugen Ein/Aus Verkehr wie Exponential Objekte, mit dem Unterschied, daß hier die “ein” und “aus” Zeiten einer Paretoverteilung entnommen werden. Sie dienen der Nachbildung des sich durch Überlappung von unabhängigen Ein/Aus-Quellen ergebenden selbstähnlichen Web-Verkehrs am Internet. Zusätzlich zu den Parametern des Exponential Objekts hat dieses Objekt noch das Attribut **shape_,** mit dem der shape Parameter der Pareto-Verteilung festgelegt wird.

Application/Traffic/CBR
CBR Objekte generieren Pakete mit einer konstanten Bitrate (“Constant Bit Rate”). Die Rate kann entweder direkt als Rate oder über das Zeitintervall zwischen den Paketen spezifiziert werden. Konfigurationsparameter sind:

- **PacketSize_** – konstante Größe erzeugter Pakete
- **rate_** – Senderate
- **interval_** – Zeitintervall zwischen Paketen
- **random_** – soll zufälliges Rauschen verwendet werden, um die Sendezeiten von Paketen schwanken zu lassen? defaultmäßig deaktiviert.
- **maxpkts_** – maximale Anzahl Pakete, die gesendet werden sollen

Application/Traffic/Trace
Diese Objekte erzeugen Verkehr aus einem trace file. Die Methode:
`$trace attach-tracefile tfile`
hängt das Tracefile Objekt tfile an diesen Verkehrsgenerator; das Tracefile Objekt spezifiziert das trace file aus dem die Verkehrsdaten gelesen werden sollen. Mehrere Application/Traffic/Trace Objekte können mit dem gleichen Tracefile Objekt verbunden werden. Jedes Application/Traffic/Trace object wählt einen zufälligen Startpunkt im Tracefile. Dieses Objekt hat keine Konfigurationsparameter.

Es gibt zwei Arten von simulierten Anwendungen: Telnet und FTP.

Application/Telnet
TELNET Objekte erzeugen Verkehr wie folgt: falls der (einzige) Konfigurationsparameter **interval_** ungleich Null ist, werden die Zeiten zwischen Paketen einer Exponentialverteilung mit durchschnittlichem **interval_** entnommen. Falls **interval_** Null ist, werden die Zeiten zwischen Paketen mittels der "tcplib" telnet Verteilung gewählt.

Application/FTP
FTP Objekte produzieren einen durchgehenden Datenstrom, der von einem TCP Objekt gesendet werden soll. Der einzige Konfigurationsparameter ist **maxpkts** – die maximale Anzahl generierter Pakete. Das FTP Objekt hat weiters folgende zwei Methoden:

- **$ftp produce n** – bringt das FTP Objekt dazu, plötzlich n Pakete zu produzieren
- **$ftp producemore count** – bringt das FTP Objekt dazu, count weitere Pakete zu erzeugen.

Übertragen von Anwendungsdaten

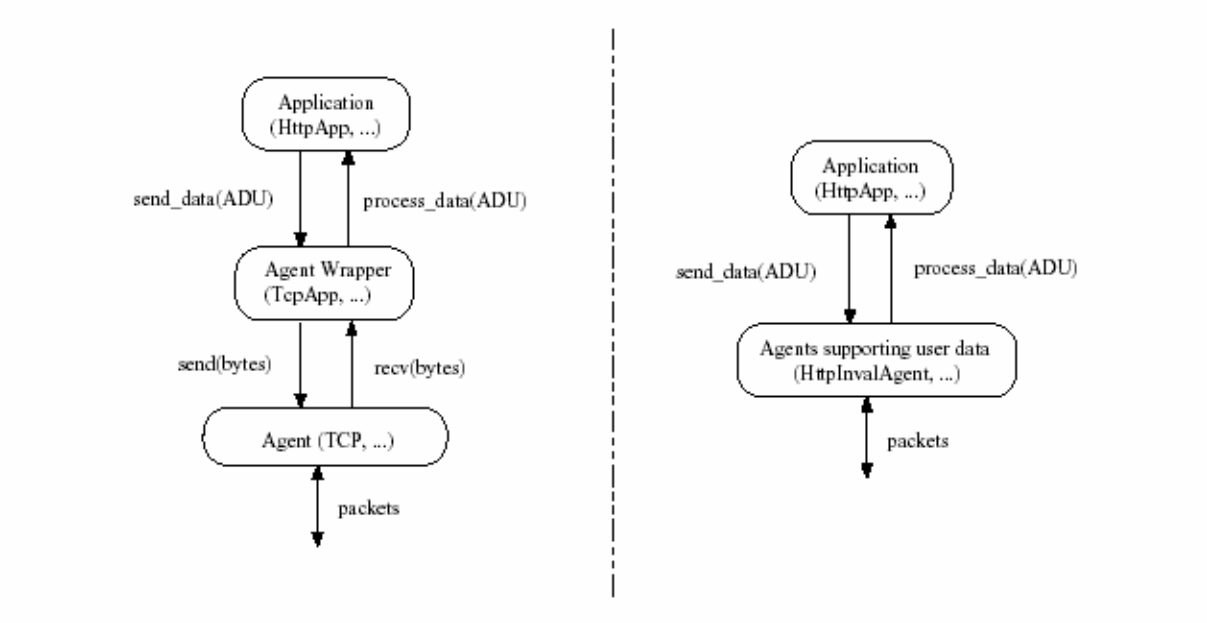
Alle Anwendungen werden als virtuelle Applikationen beschrieben. Aus diesem Grund transferieren sie normalerweise nicht selbst Daten im Simulator; die simulierten Pakete beinhalten nur die Größe und nötige Headerinformationen (Timestamps etc.). Aber ab und zu brauchen wir jedoch Anwendungen, die auch selbst Nutzlast transferieren. Ein Beispiel hierzu wäre web-caching: beim web-caching werden vom HTTP Server zu HTTP Clients "header" gesendet und diese dann "gecached". "Header" beinhalten Seitenmodifikationen, Zeitinformationen und andere Caching-Direktiven, welche für manche Cache-Konsistenz Algorithmen erforderlich bzw. wichtig sind. Anwendungsdaten ermöglichen es auch, ohne Eingriffe im Simulator selbst einfache Protokolle zu implementieren – es können etwa Pakete mit Sequenznummern versehen werden.

Um Anwendungsdaten im **ns** zu übermitteln, brauchen wir eine einheitliche Struktur, um Daten zwischen Applikationen bzw. Applikationen und Agenten austauschen zu können. Es gibt dazu drei Komponenten: eine Repräsentation der "Applicaton-level Data Unit" (ADU), ein einheitliches Interface, um Daten zwischen Applikationen auszutauschen, und zwei Mechanismen um Daten zwischen Applikationen und Transport Agents auszutauschen.

Um Daten über TCP zu übertragen, werden die Anwendungsdaten des Senders lediglich zwischengepuffert. Dann werden die Bytes gezählt, die vom Empfänger erhalten wurden. Die Klasse Application/TcpApp bietet diese Funktionalität. Ein TcpApp Objekt beinhaltet einen Pointer auf den Transport Agent. Es beinhaltet die folgenden Methoden:

`connect`
verbindet eine andere TcpApp mit der aktuellen. Diese Verbindung ist bi-direktional.

`send <nbytes> <str>`
übermittelt nbytes: die normale Größe der Applikationsdaten (es kommen noch 40 Byte Headerdaten (TCP und IP) dazu!). str: die Daten in Stringform.



Beispiel eines application-level Flusses

Kurz zusammengefasst ist die Vorgangsweise zum Übermitteln von Nutzdaten über TCP:

- zuerst SimpleTcp[16] Agents erzeugen:

```
set tcp1 [new Agent/TCP/SimpleTcp]
set tcp2 [new Agent/TCP/SimpleTcp]
$ns attach-agent $n1 $tcp1
$ns attach-agent $n2 $tcp2
$ns connect $tcp1 $tcp2
$tcp2 listen
```

Achtung: SimpleTcp realisiert eigentlich keine TCP-spezifische Funktionalität und ähnelt am ehesten einer UDP-Implementierung; wenn Pakete verloren gehen, werden sie nicht erneut geschickt.

- Dann TcpApps erzeugen und verbinden:

```
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
$app1 connect $app2
```

- Als nächstes eine Prozedur für den Empfänger definieren:

```
Application/TcpApp instproc app-recv { size } {
  global ns
  puts "[$ns now] app2 receives data $size from app1"
}
```

- Und schliesslich die eigentliche Übertragung starten:

```
$ns at 1.0 "$app1 send 100 "$app2 app-recv 100""
```

Die folgenden einfachen Beispiele sollen die Möglichkeiten illustrieren, die sich durch die Übermittlung von Nutzdaten ergeben.

[16] lt. ns manual sollten es FullTCP Agents sein – diese funktionieren jedoch nicht einwandfrei.

```
#Einfaches request response Beispiel
#Sobald eine Nachricht erhalten worden ist folgt eine Rueckantwort

set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf

#Diese Funktion wird aufgerufen, wenn etwas bei app2 ankommt
Application/TcpApp instproc app2-recv {data} {
    global app1 app2 ns

    puts "[$ns now] $data wurde erhalten"

    $app2 send 100 "$app1 app1-recv Hallo_User"
}

#Diese Funktion wird aufgerufen, wenn etwas bei app1 ankommt
Application/TcpApp instproc app1-recv {data} {
    global app1 app2 ns

    puts "[$ns now] $data kam zurueck"
}

proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exit 0
}

set n0 [$ns node]
set n1 [$ns node]

$ns duplex-link $n0 $n1 1Mb 10ms DropTail

set tcp1 [new Agent/TCP/SimpleTcp]
set tcp2 [new Agent/TCP/SimpleTcp]

$ns attach-agent $n0 $tcp1
$ns attach-agent $n1 $tcp2
$ns connect $tcp1 $tcp2
$tcp1 listen
$tcp2 listen

#Erzeugen der TCPApps und verbinden
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
$app1 connect $app2
$app2 connect $app1

$ns at 0.5 "$app1 send 100 \"$app2 app2-recv Hallo_Welt\""

$ns at 1.0 "finish"
$ns run
```

```
#Einfaches request response Beispiel mit retransmissions:
#sobald eine Nachricht erhalten worden ist folgt eine Rueckantwort;
#ein Paket geht verloren (Linkausfall), der Fehler wird durch
#ein Timeout am Sender erkannt, das Paket wird erneut geschickt


set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf


#---------------------------------------------------
#     app2: Empfaenger
#---------------------------------------------------

#Diese Funktion wird aufgerufen, wenn etwas bei app2 ankommt
proc app2-recv {data} {
    global app2 ns

    puts "[$ns now] $data wurde erhalten"
    $app2 send 100 "app1-recv Hallo_User"
}




#---------------------------------------------------
#     app1: Sender
#---------------------------------------------------

#Sendervariable: ist etwas angekommen?
set app1_received 0

#Diese Funktion wird aufgerufen, wenn etwas bei app1 ankommt
proc app1-recv {data} {
    global ns app1_received

    puts "[$ns now] $data kam zurueck"
    set app1_received 1
}

#Spezielle Send-Funktion mit timeout
proc app1-send {size data} {
    global app1 ns app1_received

    if { $app1_received == 0 } {
        $app1 send $size "app2-recv $data"
        # Timeout nach 0.5 Sekunden
        $ns at [expr [$ns now] + 0.5] "app1-send $size $data"
    } else {
        set app1_received 0
    }
}

#---------------------------------------------------
#---------------------------------------------------
```

```
proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exit 0
}

set n0 [$ns node]
set n1 [$ns node]

$ns duplex-link $n0 $n1 1Mb 10ms DropTail

set tcp1 [new Agent/TCP/SimpleTcp]
set tcp2 [new Agent/TCP/SimpleTcp]

$ns attach-agent $n0 $tcp1
$ns attach-agent $n1 $tcp2
$ns connect $tcp1 $tcp2
$tcp1 listen
$tcp2 listen

#Erzeugen der TCPApps und verbinden
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
$app1 connect $app2
$app2 connect $app1


$ns at 0.5 "app1-send 100 Hallo_Welt"

#Diese zwei Zeilen dienen zur Simulation eines verlorenen
#Paketes indem die Verbindung einfach unterbrochen wird
$ns rtmodel-at 0.51 down $n0 $n1
$ns rtmodel-at 0.53 up $n0 $n1

$ns at 2.0 "finish"
$ns run
```

```
#Dieses Beispiel zeigt, dass man mit TCPApp auch direkt
#z.B. die Bandbreite eines CBR-Agents aendern kann

set ns [new Simulator]
set nf [open out.nam w]
$ns namtrace-all $nf

set interval 0.005

#Diese Funktion bekommt einen Wert, der zur Bandbreite von
#cbr1 hinzuaddiert wird. Anschließend wird die Bandbreite ausgegeben
Application/TcpApp instproc app-recv {value} {
        global ns cbr1 interval
        set interval [expr $interval - $value]
        $cbr1 set interval_ $interval
        puts "$interval"

}

proc finish {} {
        global ns nf
        $ns flush-trace
        close $nf
        exit 0
}

set n0 [$ns node]
set n1 [$ns node]

$ns duplex-link $n0 $n1 1Mb 10ms DropTail

set cbr1 [new Agent/CBR]
set null [new Agent/Null]

$ns attach-agent $n0 $cbr1
$ns attach-agent $n1 $null
$ns connect $cbr1 $null


set tcp1 [new Agent/TCP/SimpleTcp]
set tcp2 [new Agent/TCP/SimpleTcp]

$ns attach-agent $n0 $tcp1
$ns attach-agent $n1 $tcp2
$ns connect $tcp1 $tcp2
$tcp1 listen
$tcp2 listen

#Erzeugen der TCPApps und verbinden
set app1 [new Application/TcpApp $tcp1]
set app2 [new Application/TcpApp $tcp2]
$app1 connect $app2
$app2 connect $app1

$ns at 0.0 "$cbr1 start"
$ns at 0.1  "$app1 send 100  \"$app2 app-recv 0.001\""
$ns at 0.2  "$app1 send 100  \"$app2 app-recv 0.001\""
$ns at 0.3  "$app1 send 100  \"$app2 app-recv 0.001\""
$ns at 0.4  "$app1 send 100  \"$app2 app-recv 0.001\""
$ns at 0.5 "finish"
$ns run
```

Tracing

Es gibt eine Vielzahl an Möglichkeiten, simulationsrelevante Informationen zu Sammeln, um sie zur späteren Analyse in eine Datei zu schreiben („tracing“). Eine Möglichkeit ist das aktive Abfragen bestimmter Attribute von Objekten (etwa der aktuellen Fenstergröße eines TCP-Senders) im Simulationsskript; **ns** unterstützt tracing aber auch durch explizite trace-spezifische Befehle. Wenn das Attribut des Simulators $trace AllFile definiert bzw. gesetzt ist, werden zu Links automatisch trace-Elemente hinzugefügt, welche die Pakete, die von der Queue genommen bzw. hineingelegt werden, aufzeichnen. Weiters braucht man beim „Tracen“ ein drop-Trace-Element, welches dem drophead_ Element folgt (siehe Abbildung „Aufbau eines Links“ im Abschnitt zu Links). Die folgenden Elemente speichern die Trace-Elemente:

- enqT_ : Referenz auf ein Element, welches die eingehenden Pakete aufzeichnet
- deqT_ : Referenz auf ein Element, welches die ausgehenden Pakete aufzeichnet
- drpT_ : Referenz auf ein Element, welches die gelöschten Pakete aufzeichnet
- rcvT_ : Referenz auf ein Element, welches die von einem Knoten empfangenen Pakete aufzeichnet

Die am häufigsten verwendeten Tracing Befehle von **ns** sind:

`$ns_ trace-all tracefile`
Dieser Befehl wird verwendet um Tracing in **ns** für alle Objekte zu aktivieren. Alle erhobenen Daten werden in die Datei <tracefile> geschrieben.

`$ns_ namtrace-all namtracefile`
Verrichtet im Wesentlichen das selbe wie der vorhergehende Befehl, mit dem Unterschied, dass die in die Datei <namtracefile> geschriebenen Daten wesentlich umfangreicher sind (Informationen zur graphischen Darstellung mit **nam** beinhalten).

`$ns_ flush-trace`
Dieser Befehlt leert den Trace-Puffer und wird üblicherweise kurz vor Simulationsende aufgerufen.

`$ns_ get-nam-traceall`
Gibt den namtrace-Filedescriptor zurück, der in der Simulatorinstanz-Variable namtraceAllFile_ gespeichert ist.

`$ns_ get-ns-traceall`
Ähnlich zu get-nam-traceall. Hier wird der Filedescriptor des **ns** Tracefiles zurückgegeben, der in der Simulatorinstanz traceAllFile_ gespeichert ist.

`$ns_ create-trace type file src dst optional:op`
Dieser Befehl erzeugt ein Trace Objekt vom Typ <type> zwischen den Knoten <scr> und <dst>. Die Ergebnisse werden in die Datei <file> geschrieben>. <op> ist ein Argument, das verwendet werden kann, um den Tracing-Type festzulegen. Falls <op> nicht definiert ist, wird das default Trace-Objekt `nstraces` verwendet.

`$ns_ trace-queue n1 n2 optional:file`
Dies ist eine Wrapper-Methode für create-trace. Der Befehlt richtet ein Trace-Objekt ein, um die Ereignisse im Link zwischen den Knoten <n1> und <n2> aufzuzeichnen.

`$ns_ namtrace-queue n1 n2 optional:file`

Wird verwendet um ein Trace-Objekt für namtracing am Link zwischen den Knoten <n1> und <n2> einzurichten. Die Methode ist eng verwandt und ist das Gegenstück zur Methode trace-queue.

`$ns_ drop-trace n1 n2 trace`
Macht aus einem gegebenen <trace>-Objekt ein drop-target für die Queue in Verbindung mit den Knoten <n1> und <n2>.

`$ns_ monitor-queue n1 n2 qtrace optional:sampleinterval`
Richtet einen Monitor ein, der die durchschnittliche Queue-Länge der Queue zwischen den Knoten <n1> und <n2> überwacht und aufzeichnet. Der Defaultwert von sampleinterval ist 0,1.

`$link trace-dynamics` **`ns`** `fileID`
Verfolgt die Dynamic der angegebenen Verbindung und und schreibt die Ergebnisse in die Datei fileID. **ns** ist eine Instanz des Simulators.

Im folgenden werden einige typische Trace- Methoden und Anwendungsmöglichkeiten beschrieben.

Queue Tracing

Tracing wird direkt durch bereits zuvor erwähnte Methoden von Links und Queues unterstützt. Es ist natürlich auch möglich, selbst z.B. die Länge einer Queue (Attribut des Objekts) auszulesen und den Wert in eine Datei zu schreiben:

```
proc recordQueue { file queue } {
    set ns [Simulator instance]
    #Hole die aktuelle Zeit
    set now [$ns now]
    puts $file "$now [$queue set size_]"
    #rufe die Prozedur neu auf
    $ns at [expr $now+0.5] "recordQueue $file $queue"
}
```

Dieser Prozedur muß ein Queue-Objekt übergeben werden; weiß man, daß es zwischen den Knoten n1 und n2 einen Link gibt, erhält man ein solches Objekt etwa mit dem Befehl:

```
[$ns link $bn1 $bn2] queue
```

LossMonitor

Kernstück des folgenden Anwendungsbeispiels ist die „record“-Prozedur, in der festgelegt wird, welche Daten in die Ausgabedatei geschrieben werden sollen:

```
#Erzeuge ein Simulator-Objekt
set ns [new Simulator]

#Öffne die nam-Trace-Datei
set nf [open out.nam w]
```

```
$ns namtrace-all $nf

#Öffne die Ausgabe-Datei
set f0 [open out.tr w]


#'finish'-Prozedur
proc finish {} {
  global ns nf f0
  $ns flush-trace
  close $nf
  close $f0
  exit 0
}

#'record'-Prozedur
proc record {} {
  global null0 ns f0
  #Wie viele Bytes wurden in den Traffic-Senken empfangen?
  set bw0 [$null0 set bytes_]
  #Hole die aktuelle Zeit und speicher sie in der Variable now
  set now [$ns now]
  #Berechne die Bandbreite in (MBit/s) und schreibe sie i.d. Ausgabe-Datei
  puts $f0 "$now [expr $bw0]"
  #Setzte die bytes_-Werte der Traffic-Senken auf 0 zurück
  $null0 set bytes_ 0
  #Rufe die Prozedur erneut auf
  $ns at [expr $now+1.0] "record"
}


#Erzeuge zwei Knoten
set n0 [$ns node]
set n1 [$ns node]

#Erzeuge ein Duplex-Verbindung zwischen den Knoten
$ns duplex-link $n0 $n1 1Mb 10ms DropTail

#Erzeuge einen UDP-Agenten und verknüpfe ihn mit Knoten n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

#Erzeuge eine CBR Traffic-Quelle und verknüpfe sie mit udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

#Erzeuge einen Null-Agent (Traffic-Senke) und verknüpfe ihn mit Knoten n1
set null0 [new Agent/LossMonitor]
$ns attach-agent $n1 $null0

#Verknüpfe die Traffic-Quelle mit der Traffic-Senke
$ns connect $udp0 $null0

#Lege den Zeitplan der Ereignisse für den CBR-Agent fest
$ns at 1.0 "record"
$ns at 0.5 "$cbr0 start"
$ns at 1.5 "$cbr0 stop"
$ns at 2.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

```
#Rufe die 'finish'-Prozedur nach 5 Sekunden Simulationszeit auf
$ns at 5.0 "finish"

#Starte die Simulation
$ns run
```

Ergebnis des obigen Skripts ist eine Datei mit Namen out0.tr und dem folgendem Inhalt:

```
1 20580
2 21630
3 20580
4 42000
```

Zu den Zeitpunkten in Spalte 1 wurden die in Spalte 2 dargestellten Bytes in den Traffic-Senken empfangen; für Informationen zur Visualisierung derartiger Daten sei auf Anhang B verwiesen. Mit dieser Methode können alle lesbaren Attribute mitprotokolliert werden – z.B. die aktuelle TCP-Fenstergröße am Sender. Es gibt aber auch Einschränkungen. So funktioniert obige Methode ohne Probleme mit UDP, bei der Verwendung des TCP-Protokolls stößt man allerdings schnell an die Grenzen: TCP benötigt am anderen Ende einen TCP-Sink-Agent, dem das notwendige bytes_ Attribut fehlt.

Ein etwas umfangreicheres Anwendungsbeispiel des LossMonitor Objekts findet sich in Anhang A.

Analyse einer Trace-Datei

Es gibt mehrere Möglichkeiten, Daten einer Simulation automatisch in eine Datei schreiben zu lassen. Die wichtigsten **ns**-Funktionen sind – wie bereits kurz erwähnt – namtrace-all und trace-all, wobei der wesentlichste Unterschied im Umfang der Ausgabe-Datei liegt. Sollen die Daten nicht mit **nam** visualisiert werden, ist der trace-all Befehl ausreichend.

event	time	from node	to node	pkt type	pkt size	flags	fid	src addr	dst addr	seq num	pkt id

```
r : receive (at to_node)                    src_addr : node.port (3.0)
+ : enqueue (at queue)                      dst_addr : node.port (0.0)
- : dequeue (at queue)
d : drop    (at queue)
```

```
r 1.3556 3 2 ack 40 ------- 1 3.0 0.0 15 201
+ 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
- 1.3556 2 0 ack 40 ------- 1 3.0 0.0 15 201
r 1.35576 0 2 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
d 1.35576 2 3 tcp 1000 ------- 1 0.0 3.0 29 199
+ 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
- 1.356 1 2 cbr 1000 ------- 2 1.0 3.1 157 207
```

Das Ergebnis einer Simulation wird ereignisorientiert in einer Datei abgelegt. Jede Zeile dieser Datei definiert ein solches Ereignis und mit einem Ereignis-Schlüssel:

r:	receive:	Ein Paket wurde am Zielknoten empfangen
+:	enqueue:	Ein Paket wurde in der Queue eingereiht
-:	dequeue:	Ein Paket wurde aus der Queue entfernt
d:	drop:	Ein Paket konnte von der Queue nicht aufgenommen werden

Der zweite Eintrag ist der Simulationszeitpunkt dieses Ereignisses (in Sekunden). Die nachfolgenden beiden Angaben beschreiben den Quell- und Zielknoten, also die Verbindung in der das Ereignis auftritt. Die Knoten sind fortlaufend durchnumeriert, beginnend mit `0` (Null). Die letzten beiden Informationen vor den Flags (werden als „------„ dargestellt, wenn keine Flags gesetzt wurden) sind der Paket-Typ (protokollabhängig) und die Paket-Größe (in Bytes).

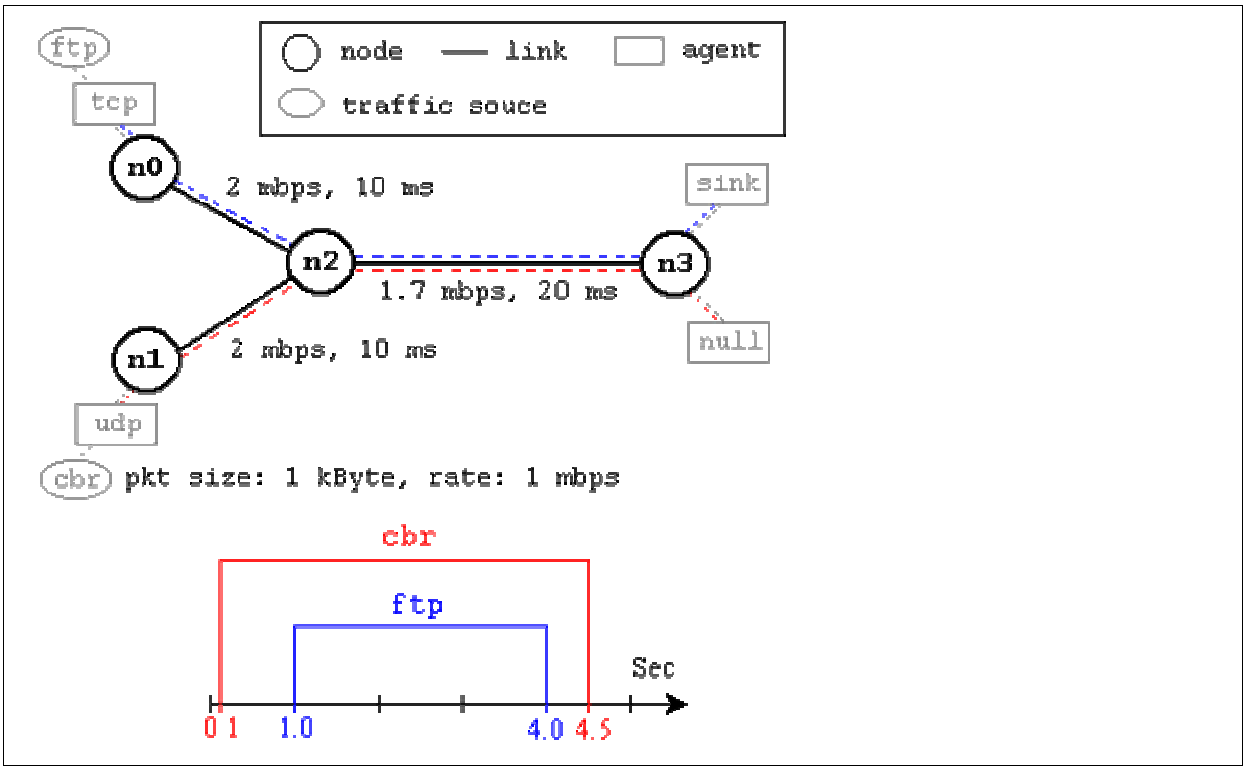
Im Moment ist im **ns** nur das „Explicit Congestion Notification“ (ECN) bit implementiert, die restlichen Flags sind noch nicht in Verwendung.

Der nächste Eintrag ist die vom Benutzer im **OTcl**-Script festgelegte „Flow ID“ (vgl. IPv6). Obwohl die ID möglicherweise nicht in der Simulation genutzt wird, kann sie doch in der darauffolgenden Analyse verwendet werden. Die Flow ID wird auch von **nam** genutzt um den Datenstrom zwischen Quell- und Zielknoten einzufärben. Die nächsten beiden Felder sind Adressangaben der Quell und Zielknoten, in der Form „`Knoten.Port`“. Im nächsten Eintrag ist die Paket-Sequenznummer des jeweiligen Netzwerk-Protokolls eingetragen (z.B. TCP - dieser Eintrag ist für UDP Implementierungen irrelevant). Das letzte Feld stellt eine eindeutige ID des Datenpaketes dar.

Die bisher besprochene Theorie soll nun an einem praktischen Beispiel erläutert werden. Betrachten wir eine Netzwerk-Testumgebung bestehend aus vier Knoten, wie aus der nachfolgenden Abbildung ersichtlich. Die bidirektionalen Verbindungen zwischen den Knoten n0 und n2 und n1 und n2 haben eine Bandbreite von 2 Mb/s und eine Verzögerung von 10 ms. Die Verbindung zwischen n2 und n3 hat eine Bandbreite von 1.7 Mb/s und eine Verzögerung von 20 ms. Jeder Knoten benutzt eine DropTail-Queue, deren maximale Größe 10 beträgt.

Ein TCP-Agent ist am Knoten n0 befestigt und eine Verbindung besteht zu einem an n3 hängenden TCP-„Sink“-Agent. Die per Standardeinstellung festgelegte maximale Größe eines von einem TCP-Agent versendbaren Paketes beträgt 1 kB. Ein TCP-„Sink“-Agent generiert und sendet ACK-Pakete zum TCP-„Sender“-Agent und gibt empfangene Pakete frei. Ein an n1 hängender UDP-Agent ist mit einem an n3 befestigten „Null“-Agent verbunden, welcher empfangene Pakete frei gibt.

Jeweils ein FTP- und ein CBR-Traffic-Generator sind mit den passenden Agenten verknüpft.



Netzwerk-Testumgebung

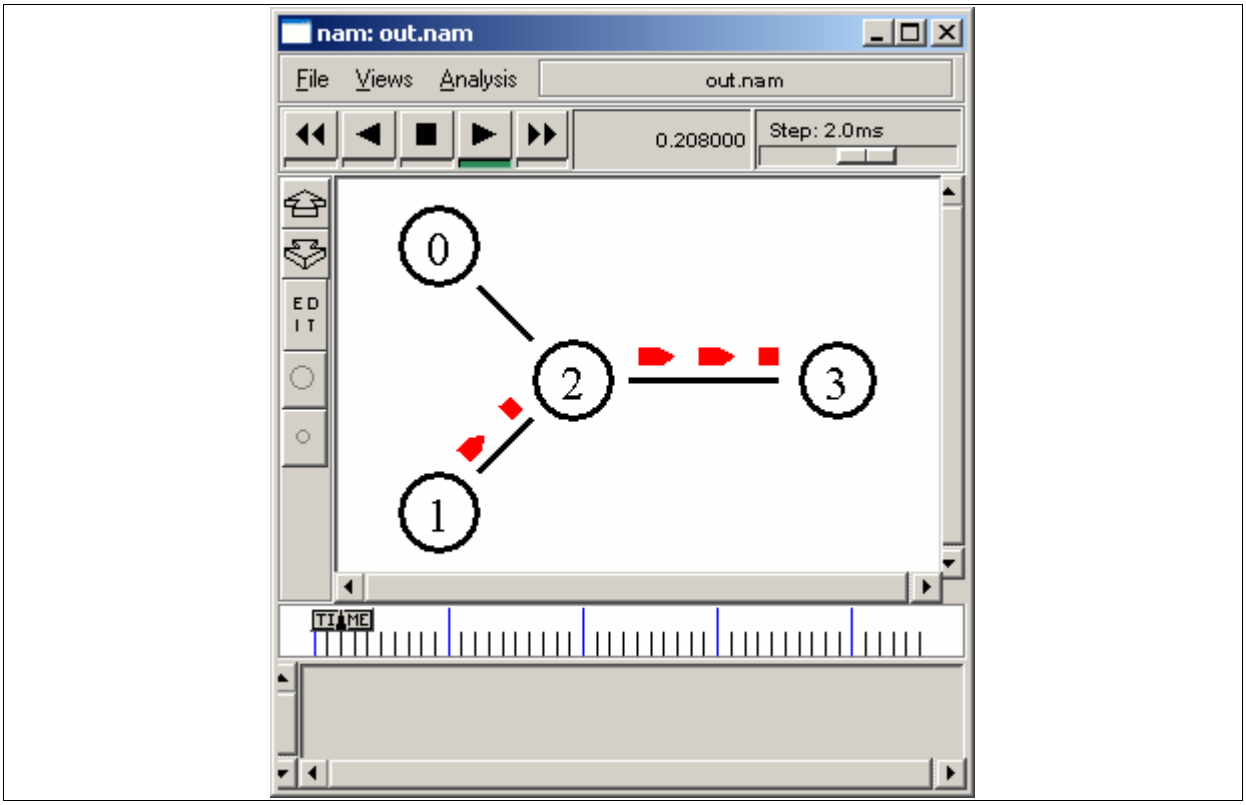
Beispiel-Skript:

```
# Erzeuge ein neues Simulator-Objekt
set ns [new Simulator]

# Lege verschiedene Farben für den Datenfluss fest (für NAM)
$ns color 1 Blue
$ns color 2 Red

# Öffne das NAM Trace-File
set nf [open out.nam w]
$ns namtrace-all $nf

# Definiere eine ‘finish’-Prozedur
proc finish {} {
  global ns nf
  $ns flush-trace
  # Schließe die NAM Trace-Datei
  close $nf
  # Wende NAM auf die Trace-Datei an (Unix; unter Windows auskommentieren!)
  exec nam out.nam &
  exit 0
}
# Erzeuge vier Knoten
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]

# Verbinde die Knoten untereinander
$ns duplex-link $n0 $n2 2Mb 10ms DropTail
$ns duplex-link $n1 $n2 2Mb 10ms DropTail
```

```
$ns duplex-link $n2 $n3 1.7Mb 20ms DropTail

# Setze die Größe der Queue der Verbindung (n2-n3) auf 10
$ns queue-limit $n2 $n3 10

# Benutzerdefinierte Anordnung der Knoten (für NAM)
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right

# Beobachte die Queue der Verbindung (n2-n3) (für NAM)
$ns duplex-link-op $n2 $n3 queuePos 0.5

# Richte eine TCP-Verbindung ein
set tcp [new Agent/TCP]
$tcp set class_ 2
$ns attach-agent $n0 $tcp
set sink [new Agent/TCPSink]
$ns attach-agent $n3 $sink
$ns connect $tcp $sink
$tcp set fid_ 1

# Richte eine FTP-Verbindung über TCP ein
set ftp [new Application/FTP]
$ftp attach-agent $tcp
$ftp set type_ FTP

# Richte eine UDP-Verbindung ein
set udp [new Agent/UDP]
$ns attach-agent $n1 $udp
set null [new Agent/Null]
$ns attach-agent $n3 $null
$ns connect $udp $null
$udp set fid_ 2

# Richte eine CBR-Verbindung über UDP ein
set cbr [new Application/Traffic/CBR]
$cbr attach-agent $udp
$cbr set type_ CBR
$cbr set packet_size_ 1000
$cbr set rate_ 1mb
$cbr set random_ false

# Erstelle einen Zeitplan für die CBR- und FTP-Agenten
$ns at 0.1 "$cbr start"
$ns at 1.0 "$ftp start"
$ns at 4.0 "$ftp stop"
$ns at 4.5 "$cbr stop"

# Trenne TCP und „Sink“-Agenten (nicht wirklich notwendig)
$ns at 4.5 "$ns detach-agent $n0 $tcp ; $ns detach-agent $n3 $sink"

# Rufe die ‘finish’-Prozedur nach fünf Sekunden Simulationszeit auf
$ns at 5.0 "finish"

# Gib die Größe eines CBR-Paketes und das Intervall aus
puts "CBR packet size = [$cbr set packet_size_]"
puts "CBR interval = [$cbr set interval_]"

# Starte die Simulation
$ns run
```

Ergebnis der Beispiel-Simulation

Das Script wird mit dem Befehl `ns dateiname.tcl` gestartet. Das Ergebnis der Simulation ist ein 712 kB großes File mit Namen `out.nam`. Dieses kann mit dem Befehl `nam out.nam` visualisiert werden.



nam Ausgabe

Für diesen Abschnitt allerdings wesentlich interessanter ist allerdings die Analyse der vorliegenden Trace-Datei. Dazu werden im Beispiel-Script die Zeilen

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

durch die Zeilen

```
set nf [open out.tr w]
$ns trace-all $tf
```

ersetzt. Es empfiehlt sich weiters den Namen der Ausgabe-Datei entsprechend den geänderten Einstellungen auf `out.tr` zu ändern. Zusätzlich wird die „finish“-Prozedur noch wie folgt angepasst:

```
proc finish {} {
  global ns tf
  $ns flush-trace
  close $tf
  # exec nam out.nam &
  exit 0
}
```

Das Skript wird wie gewohnt mit `ns datei.tcl` gestartet. Die Simulation generiert ein Tracefile mit dem Namen „`out.tr`“, das für die nachfolgende Analyse verwendet werden soll. Die ersten zehn Zeilen aus der resultierenden Trace-Datei:

```
+ 0.100 1 2 cbr 1000 ------- 2 1.0 3.1 0 0
- 0.100 1 2 cbr 1000 ------- 2 1.0 3.1 0 0
+ 0.108 1 2 cbr 1000 ------- 2 1.0 3.1 1 1
- 0.108 1 2 cbr 1000 ------- 2 1.0 3.1 1 1
r 0.114 1 2 cbr 1000 ------- 2 1.0 3.1 0 0
+ 0.114 2 3 cbr 1000 ------- 2 1.0 3.1 0 0
- 0.114 2 3 cbr 1000 ------- 2 1.0 3.1 0 0
+ 0.116 1 2 cbr 1000 ------- 2 1.0 3.1 2 2
- 0.116 1 2 cbr 1000 ------- 2 1.0 3.1 2 2
r 0.122 1 2 cbr 1000 ------- 2 1.0 3.1 1 1

[...]
```

Interessant für die Auswertung sind vor allem die ersten sechs Spalten, deren Bedeutung hier nochmals zusammengefasst werden soll:

Spalte 1: Ereignis (siehe Punkt X.X).
Spalte 2: Zeitpunkt des Ereignisses in Sekunden.
Spalte 3: ID des Quell-Knoten, von dem aus die Pakete verschickt werden.
Spalte 4: ID des Ziel-Knoten, der die verschickten Pakete empfängt.
Spalte 5: Paket-Typ (ACK, CBR, TCP, …).
Spalte 6: Größe eines verschickten Paketes in Bytes.

Die nachfolgenden Spalten sind in unserem Beispiel vernachlässigbar.

Es dürfte offensichtlich sein, dass die Visualisierung dieser Daten etwas schwieriger ist als bei den Daten, die wir im LossMonitor Beispiel erhalten haben. Zwei einfache Perl-scripts zur entsprechenden Aufbereitung von **ns** trace files werden in Anhang C beschrieben.

Anhang A: ns Schnellstart für Ungeduldige

Dieser Teil der Dokumentation bietet einen kurzen Überblick über ein paar Funktionen von **ns** sowie deren Benutzung; viele Details werden an dieser Stelle ausgespart. Als Basis diente das bekannte „Marc Greis Tutorial“ (http://www.isi.edu/nsnam/ns/tutorial/); von dort kann man sich auch den Code in jedem Entwicklungsstadium herunterladen.

Wie beginnen wir?

Als erstes empfiehlt es sich, eine Vorlage zu schreiben, die in den anschließenden Programmen als Ausgangsbasis verwendet werden kann. Das **Tcl**-script kann in jedem Texteditor geschrieben werden. Wichtig ist die Datei-Endung „.**tcl**“. Im Texteditor geben wir nun folgende Zeilen ein:

Schritt 1: erstellen eines Simulatorobjektes

```
set ns [new Simulator]
```

Schritt 2: öffnen einer Datei zum schreiben der Trace-Daten

```
set nf [open out.nam w]
$ns namtrace-all $nf
```

Die erste Zeile öffnet und erstellt eine Datei mit Namen out.nam in welches die Daten geschrieben werden. Der File-Zugriff wird an „nf“ übergeben. In der 2. Zeile teilen wir dem Simulator mit, dass alle relevanten Daten in die oben geöffnete Datei geschrieben werden.

Schritt 3: Hinzufügen einer finish Prozedur, die das tracefile schließt und **nam** startet.

```
proc finish {} {
    global ns nf
    $ns flush-trace
    close $nf
    exec nam out.nam & # unter Windows auskommentieren
    exit 0
}
```

Unter Windows muss **nam** durch den Befehl *nam out.nam* separat gestartet werden.

Schritt 4: Die darauf folgende Zeile teilt dem Simulator mit, dass er die finish procedur nach 5 sec aufrufen soll.

```
$ns at 5.0 „finish“
```

Schritt 5: Die letzte Zeile startet die Simulation

```
$ns run
```

Dieses Script kann nun zum Beispiel unter dem Namen template.tcl gespeichert werden. Wenn man allerdings das File mit „ **ns** template.tcl“ ausführt, wird man eine Fehlermeldung bekommen „**nam**: empty tracefile out.nam“ Diesen Fehler bekommen wir, weil noch keine Objekte erzeugt wurden.

Zwei Knoten und ein Link

In diesem Abschnitt werden wir eine simple Topologie erstellen, welche 2 Knoten besitzt, die über einen Link verbunden sind.

Schritt 6: Die folgenden 2 Zeilen definieren die 2 Knoten

set n0 [$ns node]
set n1 [$ns node]

(Anmerkung: man sollte die 2 Zeilen vor der Zeile *$ns run* bzw. *$ns at 5.0 „finish“* einfügen)

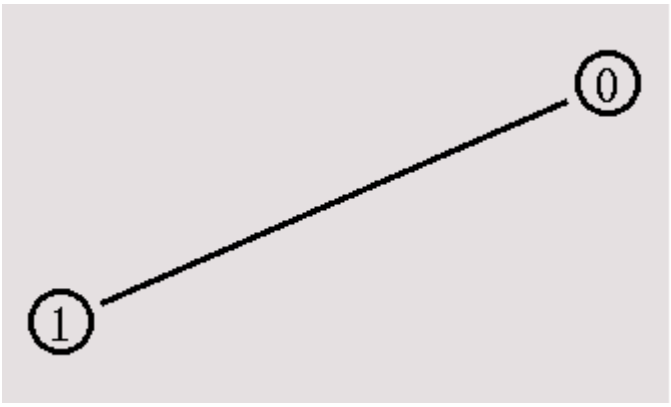
Ein neues Knotenobjekt wird mit dem Kommando *$ns node* erzeugt. Der obige Code erstellt 2 Knoten und übergibt den Zugriff an n0 und n1.

Schritt 7: Die nächste Zeile verbindet die 2 Knoten

$ns duplex-link $n0 $n1 1Mb 10ms DropTail

Diese Zeile teilt dem Simulator Objekt mit, dass die Knoten n0 und n1 mit einem „duplex“ (= bidirektionalen) Link welcher eine Bandbreite von 1 Mbit/s und eine Signallaufzeit von 10 Millisekunden sowie am Eingang eine DropTail-Queue aufweist, vebunden werden sollen. Eine „DropTail“ Queue ist eine einfache FIFO („First In, First Out“) Warteschlange. Queues sind normalerweise als Puffer in Routern realisiert – auffallenderweise werden sie in **ns** zur Vereinfachung als Teil eines Links modelliert.

Nun kann das File gesichert und mit dem Kommando *ns example1.tcl* gestartet werden .
nam wird dadurch automatisch gestartet, und folgendes Resultat sollte sichtbar sein.



Ausgabe von **nam**

Senden von Daten

Das obige Beispiel ist noch nicht sehr aufregend: man sieht nur die Topologie, doch nichts passiert. Im nächsten Schritt werden wir Daten vom Knoten n0 zum Knoten n1 senden. In **ns** werden Daten immer von einem Agenten zum anderen gesendet, daher müssen wir nun ein Agenten-Objekt erstellen, welches Daten vom Knoten n0 sendet und ein Agent-Objekt erstellen, welches diese Daten am Knoten n1 empfängt.

```
#Create a UDP agent and attach it to node n0
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

# Create a CBR traffic source and attach it to udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0
```

Diese Zeilen erstellen einen UDP Agent welcher mit dem Knoten n0 verbunden wird, dann wird ein CBR (Constant Bit Rate) Verkehrsgenerator zum UDP-agent hinzugefügt. Die Paketgröße wird dabei (als Atttribut des cbr0 Objekts) auf 500 Bytes gesetzt und ein Paket wird alle 0.005 Sekunden geschickt. Alle relevanten Parameter für jeden Agenten Typ werden später behandelt.

Als nächstes wird ein Null-Agent erzeugt welcher als Traffic-Senke fungiert und an den Knoten n1 angehängt wird.

```
set null0 [new Agent/Null]
$ns attach-agent $n1 $null0
```

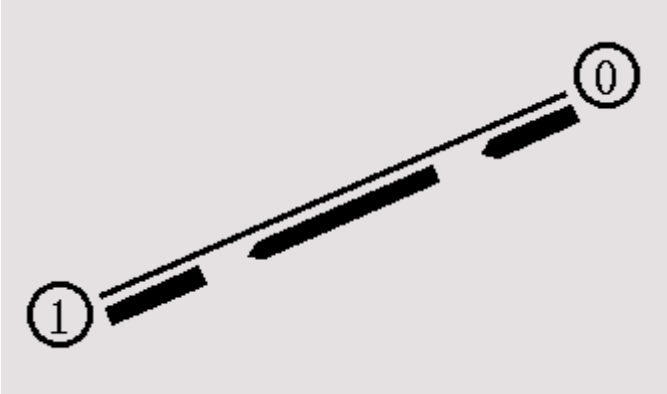
Nun müssen diese 2 Agents miteinander verbunden werden:

```
$ns connect $udp0 $null0
```

Als letztes müssen wir dem CBR-Agent mitteilen, wann und wie lange er die Daten senden soll. Diese 2 Zeilen sollten dazu vor „$ns at 5.0 „finish““ eingefügt werden:

```
$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

Jetzt kann die Datei gespeichert und wiederholt gestartet werden, und jetzt sieht man, dass etwas Leben in die Sache gekommen ist. Nach 0.5 Sekunden werden die Daten gesendet.



Drei Hosts und ein Router

In diesem Abschnitt werden wir eine Topologie definieren, welche 5 Knoten besitzt, und in der ein Knoten als Router fungiert. Dieser Router hat die Aufgabe, die Daten an die restlichen Knoten weiterzuleiten. Damit soll gezeigt werden, wie auch eine Queue simuliert werden kann.

Schritt 1: Nehmen wir wieder unser Gerüst zur Hand (neues Script erstellen) und fügen folgende Zeilen ein:

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
```

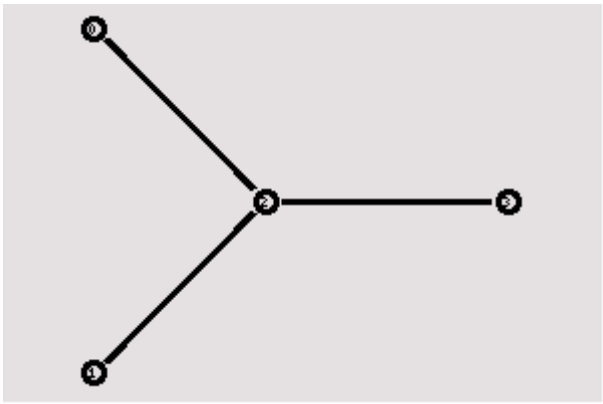
Schritt 2: Der folgende Abschnitt kreiert 2 Duplex-Links zwischen jeweils 2 Knoten:

```
$ns duplex-link $n0 $n2 1Mb 10ms DropTail
$ns duplex-link $n1 $n2 1Mb 10ms DropTail
$ns duplex-link $n3 $n2 1Mb 10ms DropTail
```

Schritt 3: Wenn dieses Script jetzt gespeichert und gestartet wird, muss man leider feststellen, dass das Layout etwas unglücklich ausgefallen ist. Um mehr Kontrolle über das Layout zu bekommen, werden auch die nächsten 3 Zeilen in unser Script hinzugefügt:

```
$ns duplex-link-op $n0 $n2 orient right-down
$ns duplex-link-op $n1 $n2 orient right-up
$ns duplex-link-op $n2 $n3 orient right
```

Man wird die obigen Zeilen besser verstehen wenn man die Topologie mit **nam** ansieht:



Schritt 4: Nun erstellen wir 2 UDP-Agents mit einer CBR Traffic-Quelle und fügen diese dann den Knoten n0 und n1 zu. Dann erstellen wir noch einen Null-Agenten und fügen diesen dem Knoten 3 zu.

```
#Erstelle einen UDP Agent und füge diesem dem Knoten n0 hinzu.
set udp0 [new Agent/UDP]
$ns attach-agent $n0 $udp0

#Erzeuge eine CBR traffic source und füge diese dem udp0 (Agenten) hinzu.
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

# Erstelle einen UDP Agent und füge diesem dem Knoten n1 hinzu.
set udp1 [new Agent/UDP]
$ns attach-agent $n1 $udp1

#Erzeuge eine CBR traffic source und Füge diese dem udp0 (Agenten)hinzu.
set cbr1 [new Application/Traffic/CBR]
$cbr1 set packetSize_ 500
$cbr1 set interval_ 0.005
$cbr1 attach-agent $udp1

set null0 [new Agent/Null]
$ns attach-agent $n3 $null0
```

Schritt 5: Die beiden CBR-Agents müssen mit dem Null-Agenten verbunden werden.

```
$ns connect $udp0 $null0
$ns connect $udp1 $null0
```

Schritt 6: Wir möchten, dass der erste CBR-Agent nach 0.5 sec zum Senden beginnt und nach 4.5 sec wieder stoppt. Der zweite CBR-Agent soll nach einer Sekunde zu senden beginnen und zum Zeitpunkt 4 wieder aufhören.

```
$ns at 0.5 "$cbr0 start"
$ns at 1.0 "$cbr1 start"
$ns at 4.0 "$cbr1 stop"
$ns at 4.5 "$cbr0 stop"
```

Schritt 7: Wenn man das Skript jetzt startet, wird man feststellen, dass wesentlich mehr Verkehr zwischen den Links n0 und n2 bzw. n1 und n2 als zwischen den Links n2 und n3 auftritt. Eine simple Berechnung zeigt uns: wir senden 200 Pakete pro Sekunde an jeden der

ersten 2 Links mit einer Paketgröße von 500 Bytes – das ergibt eine Bandbreite von 0.8 Mb pro Sekunde für die Links zwischen n0 und n2 bzw. n1 und n2; das ist eine Gesamtbandbreite von 1,6 Mb pro Sekunde, aber der Link zwischen n2 und n3 hat nur eine Kapazität von 1Mb pro Sekunde – was zur Folge hat, dass einige Pakete verloren gehen. Aber welche gehen verloren?

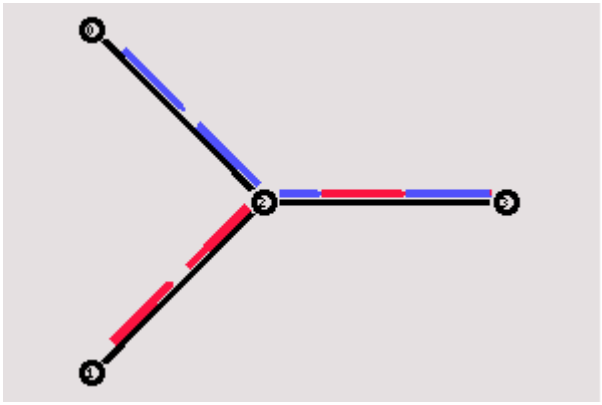
Beide Datenströme sind in **nam** schwarz. Es ergibt sich somit nur ein einziger möglicher Weg diese Pakete zu unterscheiden: Klicken auf den Datenstrom.

Schritt 8: Um die Unterscheidung verschiedener Datenflüsse zu vereinfachen, sollten wir unsere Datenflüsse markieren. Das geschieht durch Hinzufügen der folgenden 2 Zeilen zu unseren CBR-Agent Definitionen:

```
$udp0 set class_ 1
$udp1 set class_ 2
```

Die nächsten 2 Zeilen sollten eher am Beginn des **Tcl**-Skript stehen, denn sie weisen jeder Datenfluss-ID eine Farbe zu.

```
$ns color 1 Blue
$ns color 2 Red
```



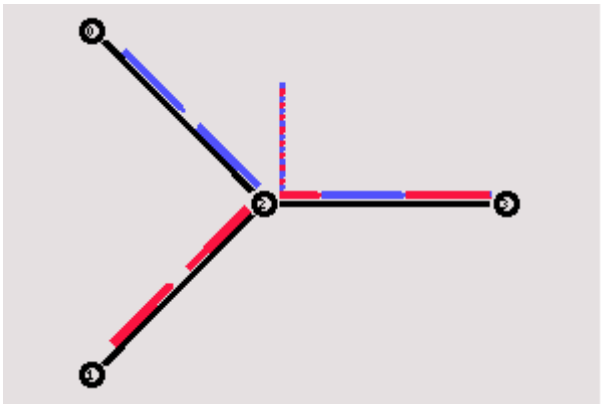
Jetzt kann das Script wiederholt gestartet werden. Siehe Bild!

Wenn man den Link zwischen den Knoten n2 und n3 für eine Weile beobachtet, kann man feststellen, dass nach einer gewissen Zeit die Verteilung der blauen und roten Datenpakete nicht gerade „gerecht“ ist. Im nächsten Schritt werden wir zeigen, wie man sich so einen Knoten bzw. den Link zwischen Knoten n2 und n3 genau anschauen kann.

Schritt 9: Man braucht nur eine einzige Zeile zu unserem Code hinzuzufügen, um die Queue am Eingang des Links zwischen n2 und n3 zu überwachen:

```
$ns duplex-link-op $n2 $n3 queuePos 0.5
```

Startet man den **ns** nochmals, dann sollte man anfangs das gleiche Bild vor sich haben wie oben – nach einiger Zeit jedoch kann man die Pakete in der Queue sehen, und etwas später sieht man wie die einzelnen Pakete verloren gehen, da die Queue überläuft:



Um das Verhältnis auf der Queue auszugleichen sollte man noch folgende Zeilen hinzufügen:

```
$ns duplex-link $n3 $n2 1Mb 10ms SFQ
```

(SFQ = „Stochastic Fair Queueing“)
Nach nochmaligen starten wird man sehen dass das Verhalten in der Queue nun fair ist.

Damit ist unser Beispiel fertig. Für den vollständigen Code sei erneut auf das „Marc Greis Tutorial“ (http://www.isi.edu/nsnam/ns/tutorial/) verwiesen.

Erstellen einer größeren Topologie

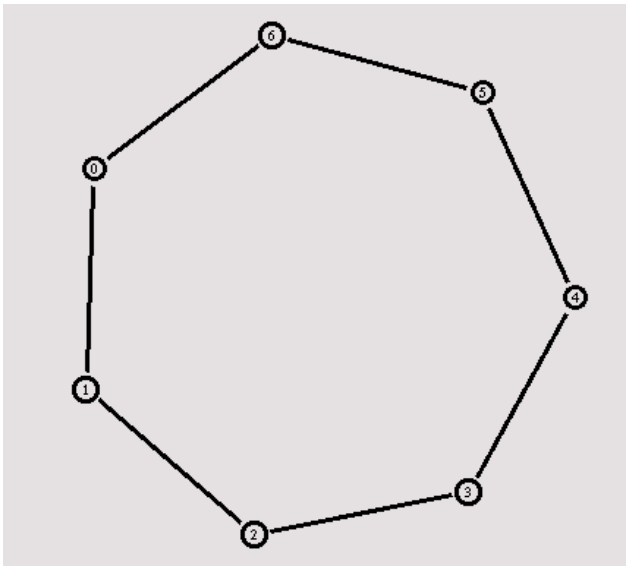
Anmerkung:
Für die folgendenBeispiele kann unsere Vorlage („template.tcl“) verwendet werden.

Zuerst muss wie immer die Topologie erzeugt werden. Es gibt mehrer Möglichkeiten, um bequem eine größere Topologie darstellen zu können – folgender Code erzeugt 7 Knoten und speichert diese in einem Array n():

```
for {set i 0} {$i < 7} {incr i} {
  set n($i) [$ns node]
}
```

Arrays verhalten sich in **Tcl** wie andere Variablen und müssen daher nicht deklariert werden. Als nächsten Schritt verbinden wir die Knoten zu einer kreisförmigen Topologie; siehe folgenden Code und Bild:

```
for {set i 0} {$i < 7} {incr i} {
  $ns duplex-link $n($i) $n([expr ($i+1)%7]) 1Mb 10ms DropTail
}
```



Verbindungsausfall

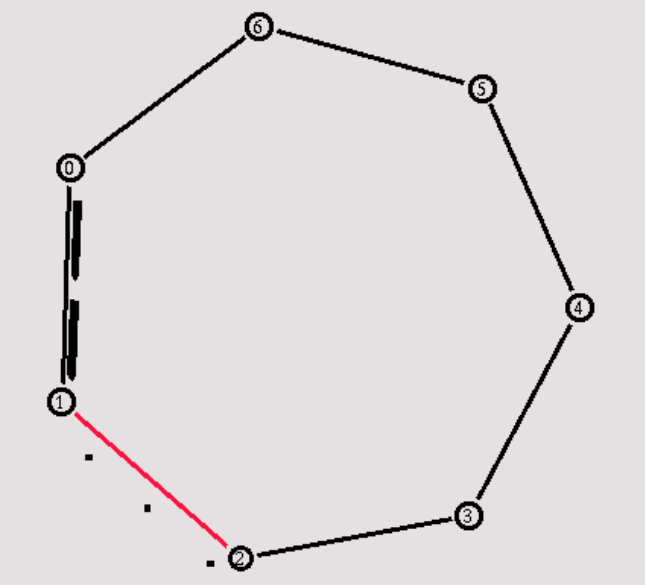
Der nächste Schritt ist es, Daten vom Knoten n0 bis zum n3 zu schicken:

```
#Erzeuge einen UDP Agenten und verbinde ihn mit dem Knoten n(0)
set udp0 [new Agent/UDP]
$ns attach-agent $n(0) $udp0

#Erzeuge eine CBR traffic source verbinde sie mit udp0
set cbr0 [new Application/Traffic/CBR]
$cbr0 set packetSize_ 500
$cbr0 set interval_ 0.005
$cbr0 attach-agent $udp0

set null0 [new Agent/Null]
$ns attach-agent $n(3) $null0

$ns connect $udp0 $null0

$ns at 0.5 "$cbr0 start"
$ns at 4.5 "$cbr0 stop"
```

Beim Starten des Skriptes sieht man, dass der Traffic den kürzesten Weg zwischen Knoten 0 und 3 nimmt => über 1 und 2.
Nun machen wir das Ganze etwas interessanter. Wir lassen den Link zwischen 1 und 2 kurzzeitig offline gehen.

```
$ns rtmodel-at 1.0 down $n(1) $n(2)
$ns rtmodel-at 2.0 up $n(1) $n(2)
```

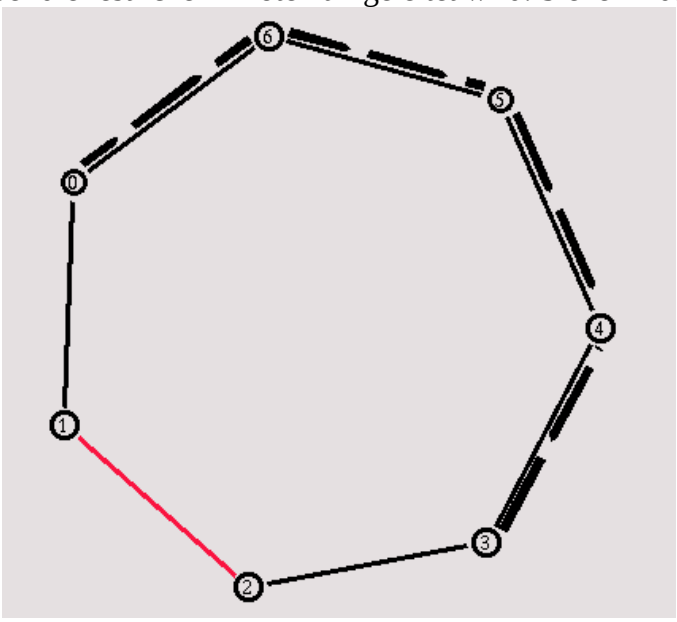
Wird der Simulator nochmals gestartet, dann sieht man dass zwischen Sekunde 1 und 2 der Link offline geht und alle Daten die von Knoten 0 gesendet werden gehen verloren.



Als nächstes zeigen wir, wie dynamisches Routing dieses Problem lösen kann. Nun wird folgende Zeile am Beginn nach dem Erstellen des Simulator-Objektes hinzugefügt:

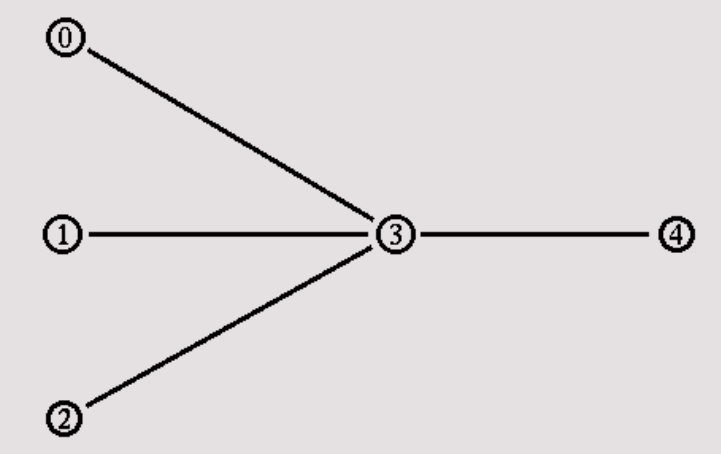
```
$ns rtproto DV
```

Nach erneutem Starten wird man sehen, dass nach dem Ausfall der Verbindung zwischen 1 und 2 der Verkehr über die restlichen Knoten umgeleitet wird. Siehe Bild.



Erstellen von Outputfiles für xGraph

Ein Teil des **ns** – allinone Paketes ist das Diagramm-Visualisierungstool xGraph. In diesem Abschnitt werden wir einen einfachen Weg zeigen, um ein von xGraph verwendbares Outputfile in **tcl** Scripts zu erstellen. Weiters werden wir sehen, wie man Verkehrsgeneratoren verwendet.



Der folgende Code sollte ohne weitere Erklärungen verständlich sein:

```
set n0 [$ns node]
set n1 [$ns node]
set n2 [$ns node]
set n3 [$ns node]
set n4 [$ns node]

$ns duplex-link $n0 $n3 1Mb 100ms DropTail
$ns duplex-link $n1 $n3 1Mb 100ms DropTail
$ns duplex-link $n2 $n3 1Mb 100ms DropTail
$ns duplex-link $n3 $n4 1Mb 100ms DropTail
```

Wir verbinden die Traffic-Sources (n3) mit dem Knoten n0, n1 und n2 – aber zuerst schreiben wir eine Prozedur, welche uns erleichtern wird, die Verkehrsquellen und Generatoren zu den Knoten hinzuzufügen.

```
proc attach-expoo-traffic { node sink size burst idle rate } {
        #Instanz des Simulators (ns)
        set ns [Simulator instance]

        #Erstelle einen UDP Agenten und verbinde ihn mit dem Knoten
        set source [new Agent/UDP]
        $ns attach-agent $node $source

        #Erstelle einen  Expoo traffic agent und setze die
        #Konfigurationsparameter
        set traffic [new Application/Traffic/Exponential]
        $traffic set packet-size $size
        $traffic set burst-time $burst
        $traffic set idle-time $idle
        $traffic set rate $rate

         #Verbinde die  traffic source mit dem  traffic Generator
         $traffic attach-agent $source
        #Verbinde die Quelle und die Senke.
        $ns connect $source $sink
        return $traffic
}
```

Diese Prozedur schaut wesentlich gefährlicher aus als sie eigentlich ist. Sie besitzt 6 Argumente: Einen Knoten, eine vorerzeugte Datensenke, eine Paketgröße für die Verkehrsquelle, Burst-Time, eine IDLE-Time und die Peak-Rate. Auf Details wird später eingegangen.

Als erstes erzeugt die Prozedur eine Verkehrsquelle und verbindet diese mit dem Knoten, danach erzeugt sie ein Traffic/Expo–Objekt, setzt dessen Konfigurationsparameter und fügt diesen der Verkehrsquelle hinzu bevor die Source und die Sink verbunden werden. Zum Schluss gibt die Prozedur einen Zugriff auf die Traffic-Source zurück. Diese Prozedur ist ein gutes Beispiel dafür, wie man Verkehrsquellen mit verschiedenen Knoten verbinden kann. Als nächstes verwenden wir diese Prozedur, um Verkehrsquellen mit verschiedenen Peak-Rates mit n0, n1 und n2 zu verbinden. Diese verbinden wir dann später mit den 3 Verkehrsquellen über n4, welche zuerst kreiert werden müssen.

```
set sink0 [new Agent/LossMonitor]
set sink1 [new Agent/LossMonitor]
set sink2 [new Agent/LossMonitor]
$ns attach-agent $n4 $sink0
$ns attach-agent $n4 $sink1
$ns attach-agent $n4 $sink2

set source0 [attach-expoo-traffic $n0 $sink0 200 2s 1s 100k]
set source1 [attach-expoo-traffic $n1 $sink1 200 2s 1s 200k]
set source2 [attach-expoo-traffic $n2 $sink2 200 2s 1s 300k]
```

In diesem Beispiel verwenden wir ein Agent/LossMonitor–Objekt als Datensenke.

Speichern der Daten im Outputfile

Als erstes müssen wir 3 Outputfiles erstellen:

```
set f0 [open out0.tr w]
set f1 [open out1.tr w]
set f2 [open out2.tr w]
```

Diese Dateien müssen an einem bestimmten Punkt geschlossen werden, dazu verwenden wir eine modifizierte finish-Prozedur:

```
proc finish {} {
        global f0 f1 f2
        #Schliesse das Output-File.
        close $f0
        close $f1
        close $f2
        #Aufrufen von xgraph um die Ergebnisse darstellen zu können.
        exec xgraph out0.tr out1.tr out2.tr -geometry 800x400 &
        exit 0
}
```

Die obige Prozedur schließt nicht nur die Outputfiles, sondern startet auch gleichzeitig den xGraph um die Daten dazustellen. Als nächstes schreiben wir die Prozedur, welche die Daten in das Outputfile schreibt:

```
proc record {} {
        global sink0 sink1 sink2 f0 f1 f2
        #Instanz des Simulators (ns)
        set ns [Simulator instance]
        #Zeit setzen (nach welcher die Prozedur wiederholt warden soll).
        set time 0.5
        #Wieviele Bytes wurden von der traffic-Senke empfangen?
        set bw0 [$sink0 set bytes_]
        set bw1 [$sink1 set bytes_]
        set bw2 [$sink2 set bytes_]
        #Die aktuelle Zeit setzen
        set now [$ns now]
        #Die Bandbreite berechnen (in MBit/s) und in die Files schreiben.
        puts $f0 "$now [expr $bw0/$time*8/1000000]"
        puts $f1 "$now [expr $bw1/$time*8/1000000]"
        puts $f2 "$now [expr $bw2/$time*8/1000000]"
        #Reset der bytes_ Werte von der traffic-Senke
        $sink0 set bytes_ 0
        $sink1 set bytes_ 0
        $sink2 set bytes_ 0
        #Neustarten der Prozedur
        $ns at [expr $now+$time] "record"
}
```

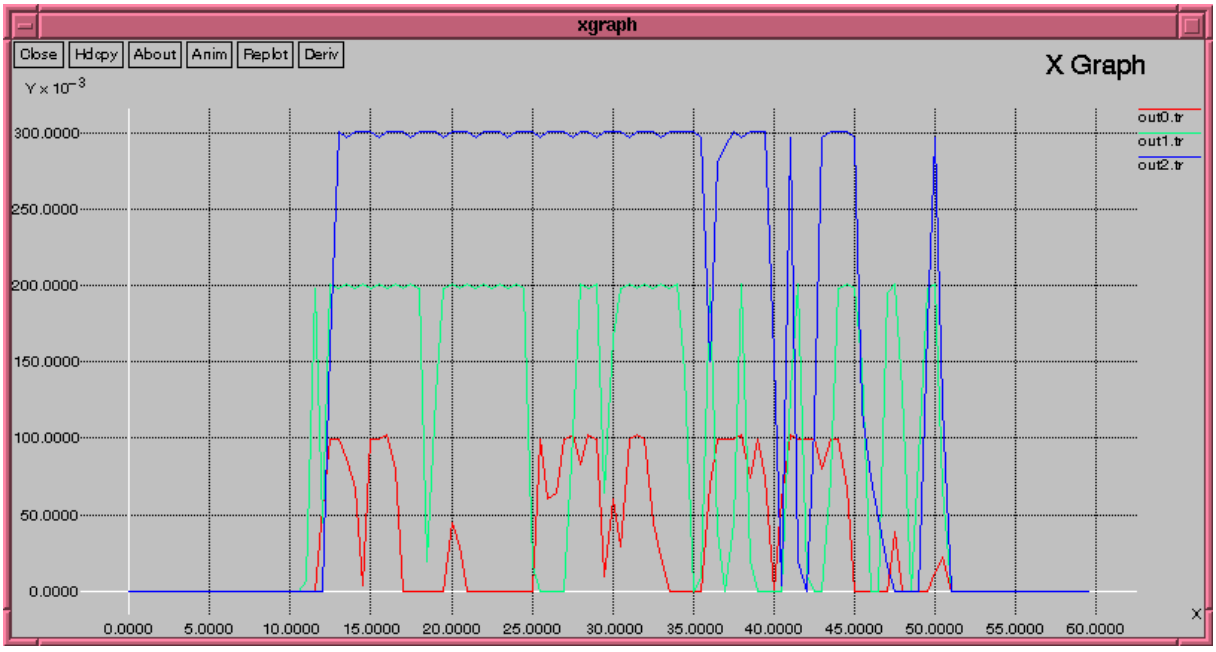
Hier wird die Anzahl der Bytes gelesen, welche von den Datensenken empfangen werden. Danach berechnet die Prozedur die Bandbreite in Mbit/s und schreibt das Ergebnis in die drei Ausgabe-Dateien zusammen mit der aktuellen Zeit bevor die bytes _value an den Datensenken zurückgesetzt werden. Man sieht hier bereits den Haken dieser Methode, um Outputfiles zu erstellen: es werden Agenten vom Typ LossMonitor benötigt; einem TCP-Empfänger (TCPSink) etwa fehlt das Attribut bytes_. Wie man in so einem Fall dennoch sinnvolle Ausgaben erzeugt, wird im Abschnitt über Tracing erklärt.

Starten der Simulation

Wir können nun folgende Ereignisse ausführen:

```
$ns at 0.0 "record"
$ns at 10.0 "$source0 start"
$ns at 10.0 "$source1 start"
$ns at 10.0 "$source2 start"
$ns at 50.0 "$source0 stop"
$ns at 50.0 "$source1 stop"
$ns at 50.0 "$source2 stop"
$ns at 60.0 "finish"

$ns run
```

Als erstes wird die record Prozedur aufgerufen und danach wird sie nach jeweils 0.5 Sekunden neu festgelegt. Dann starten alle Verkehrsquellen nach 10 Sekunden und stoppen nach 50 Sekunden. Nach 60 Sekunden wird die finish Prozedur aufgerufen. Wenn man nun die Simulation ausführt, sollte ein xGraph-Fenster wie nachfolgend erscheinen:



In diesem Fenster sieht man die dynamische Entwicklung des bei den drei Datensenken eintreffenden Verkehrs.

Anmerkung: Das Outputfile kann auch mit gnuplot (Anhang B) verwendet werden.

Anhang B: Visualisierung mit gnuplot

Was ist gnuplot?

Gnuplot ist ein interaktives Plotprogramm. Mit wenigen einfachen Befehlen können 2-dimensionale und 3-dimensionale Darstellungen von Funktionen und eingelesenen Datensätzen erzeugt werden.

gnuplot stammt von Thomas Williams, Colin Kelley und zahlreichen weiteren Programmierern, die seit 1986 das ursprünglich kleine Programm zum Allroundwerkzeug entwickelt haben. Aktuell ist zum Zeitpunkt, an dem diese Dokumentation geschrieben wurde, die Version 3.7, Patchlevel 3. Der klein geschriebene Eigenname lässt schon darauf schließen: es ist eines von zahlreichen Werkzeugen, die man in wissenschaftlichen Disziplinen zum „Plotten“ einsetzt: zum graphischen Abbilden (Visualisieren) von Datenreihen oder mathematischen Funktionen.

Visualisieren dient dem Zweck, abstraktes Zahlenmaterial, etwa Reihen von Messwerten oder z.B. eine Rohdatenmatrix aus der Umfrageforschung in eine für den Menschen leichter verständliche und handhabbare Form aufzubereiten und gleichzeitig eine Analyse vorliegender Daten zu ermöglichen und zu begleiten. Die Wurzeln von gnuplot liegen in der Unixwelt, es ist praktisch jeder gängigen modernen Linux-Distribution beigefügt, im Internet ist auch eine Version für Microsoft Windows (32-bit) verfügbar[17].

Was kann gnuplot?

gnuplot kann sowohl zweidimensionale als auch (pseudo-) dreidimensionale Darstellungen erzeugen und beherrscht polare und kartesische Koordinatensysteme mit linearer und logarithmischer Skalierung. Es verwendet analytische und parametrische Verfahren, kann die eingelesenen Rohdaten nicht nur abbilden (z.B. als Kurven, Balken oder 3D-Objekte), sondern bietet auch zusätzliche Funktionen, wie etwa das Integrieren von Fehlertermen (Streuungsmaße) in der graphischen Darstellung. Tortendiagramme sind nicht die Stärke von Gnuplot. Ebenso – soweit den Autoren bekannt – ternäre Diagramme, was für Geologen natürlich schade ist...

Wie funktioniert gnuplot?

Gnuplot ist wie gesagt ein kommandozeilen-orientiertes, interaktives Programm, d.h. beim Aufruf erscheint ein Kommandofenster mit folgendem Prompt:

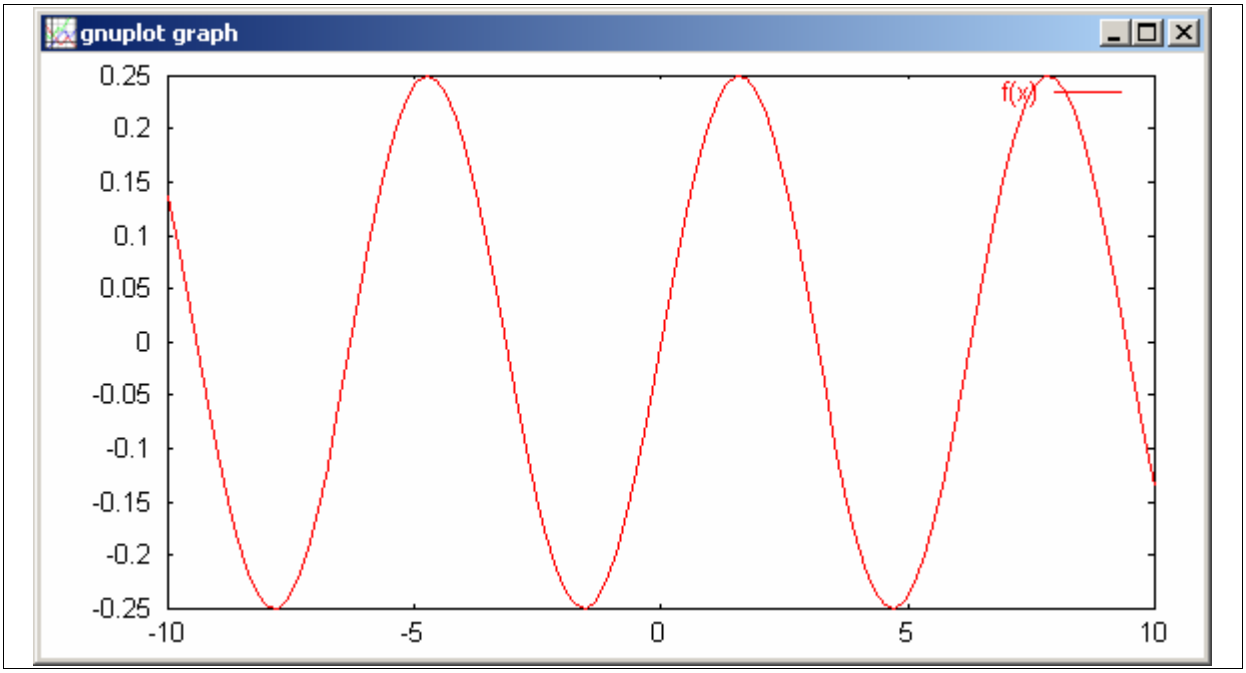
```
gnuplot>
```

das eine Eingabe verlangt, z.B. einen der Plotbefehle oder die Definition einer Funktion oder Angaben zur Formatierung einer Achse oder etwas Ähnliches. Ist eine Eingabe erfolgt, wird sie ausgeführt, was man entweder sofort bemerkt (bei einem Plotbefehl) oder was sich später auf folgende Plotbefehle auswirkt (bei Funktionsdefinitionen z.B.)

```
gnuplot> a=0.25
gnuplot> f(x)=sin(x)*a
gnuplot> plot f(x)
```

[17] ftp://ftp.gnuplot.info/pub/gnuplot/gp373w32.zip

Diese Befehlsabfolge (jeweils mit einem ENTER abgeschlossen) hätte beispielsweise das folgende Resultat:



Beispielausgabe mit gnuplot

Praktischerweise wird man diese Befehlsfolgen nicht immer wieder neue eingeben, sondern in eine Stapeldatei oder `Script-Datei` (eine einfache Textdatei, in Folgenden einfach plt-Datei genannt) schreiben, die dann aufgerufen werden kann (Befehl: `load 'Datei.plt'`) und die Befehlsabfolge in Gang setzt. So kann man eine Datei immer weiter editieren, z.B. um die Achsenlänge zu verändern, dasselbe Grafikformat auf unterschiedliche Daten loszulassen etc.

Voreinstellungen

Es gelten schon einige vordefinierte Standard-Einstellungen über den Stil der Abbildung z.B. dass die erste Grafik in rot gehalten wird etc. Angezeigt werden die gültigen Einstellungen mit dem Befehl `show all`. Diese Grundeinstellungen stehen in einer Datei namen `.gnuplot` (UNIX, LINUX), bzw. `GNUPLOT.INI` (Windows). Diese Defaults kann man natürlich überschreiben. Der prinzipielle Befehl, um Änderungen an Achsen, Label, Wertebereiche etc. vorzunehmen lautet `set`. Eine Änderung wirkt sich solange aus, bis man die Einstellungen neu festgelegt hat. Mit `reset` kehrt man zu den Standardeinstellungen zurück.

Wichtige Gnuplot-Befehle anhand von Beispielen

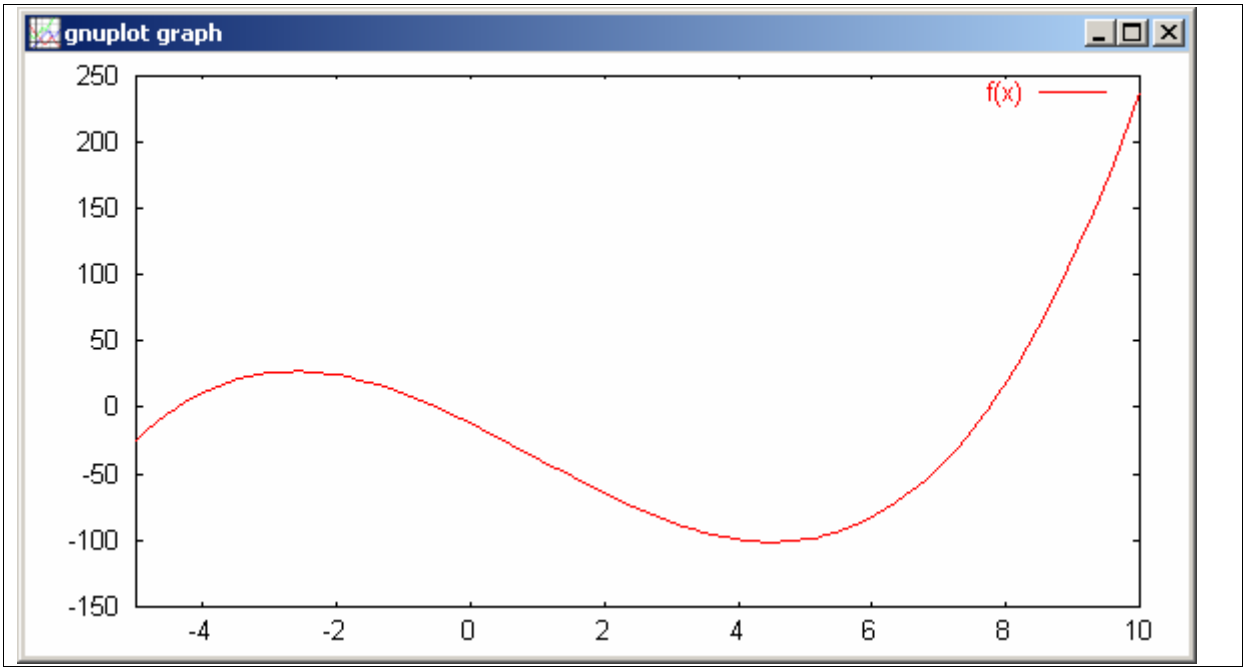
Im Folgenden werden einzelne wichtige Befehle und Funktionen vorgestellt und illustriert, was sie bewirken. Zu jedem der Befehle existiert eine detaillierte Beschreibung, die mit dem Befehl `help` abfragen kann (z.B. `help set`).

Variablen und Funktionen definieren, Achsen formatieren

Untenstehender Code kann beispielsweise in eine Datei mit namen `plot.plt` gespeichert und mit dem Befehl `load ‚plot.plt'` mit gnuplot visualisiert werden:

```
set xrange [-5:10]
set yrange [-150:250]
a=0.7
b=-2.0
c=-25
f(x)=a*x**3+b*x**2+c*x-12
plot f(x)
```

Das Ergebnis: ist das Folgende:



Variablen- und Funktionsdefinition, Achsenformatierung

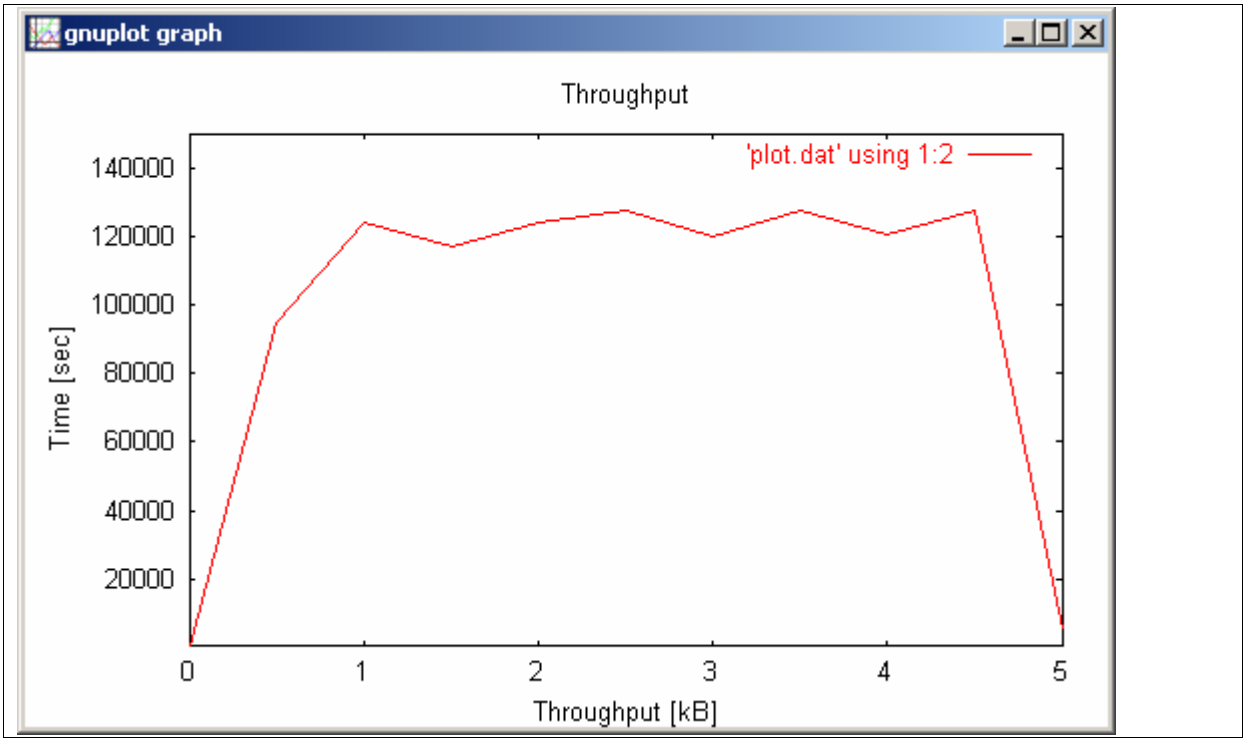
Was passiert hier? Zunächst einmal werden die darzustellenden Achsenbereiche festgelegt: x von -5 bis 10, y von -150 bis 250, dann werden einige Variablen definiert, die dann später in der Funktionsdefinition wieder aufgegriffen werden. Der Plot-Befehl `plot f(x)` stellt die Funktion dar.

Daten plotten, Achsen beschriften

Im Folgenden sollen Werte aus einer Datei in ein Koordinatensystem eingetragen und grafisch dargestellt werden. Dazu legt man eine Datei mit folgendem Inhalt an:

```
set title 'Throughput'
set xrange [0:5]
set yrange [500:150000]
set xlabel 'Throughput [kB]'
set ylabel 'Time [sec]'
plot 'plot.dat' using 1:2 with line
```

Der Aufruf des Skripts erfolgt, wie gewohnt, durch `load ‚dateiname.plt'` in gnuplot und liefert abgebildetes Ergebnis:



Rohdaten aus einer Datei

Mit `set title` wurde die Grafik beschriftet, mit `set xlabel` und `set ylabel` wurden die Achsen beschriftet. Im Unterschied zum vorigen Beispiel wird hier keine Funktion dargestellt, sondern gemessene Daten, die aus einer Datei (`input.dat`) bezogen werden. Es handelt sich um eine einfache Textdatei, in der die x/y-Wertepaare (Konzentration gegen Entfernung) als zwei Spalten (getrennt je durch ein paar Leerzeichen) stehen. So etwas könnte z.B. eine aus MS EXCEL exportierte Datei sein, oder die in eine Datei umgeleitete Ausgabe des Tracefile-Analysetools „Throughput.pl". Man kann einzelne Spalten einer Datei gegeneinander darstellen (z.B. 1. Spalte = x, 2. Spalte = y) durch den Zusatz `using 1:2`.

Sollen die Daten nicht verbunden werden, sondern als Punkte dargestellt werden, kann das Zauberwort `with` verwendet werden, der den plot-Befehl mit einem `STYLE` verbindet (für genauere Angaben: `help plot`).

Für weitere Informationen („wie erzeuge ich mit gnuplot eine postscript Ausgabedatei?", ..) sei auf http://www.duke.edu/~hpgavin/gnuplot.html verwiesen.

Anhang C: Aufbereitung von ns trace files

Das trace-file Format von **ns** ist sehr umfangreich und an Events orientiert. In der Praxis hat sich gezeigt, daß derartige Daten kaum unmittelbar mit gnuplot oder xgraph visualisierbar sind; meistens benötigt man entweder Sammeldaten bezüglich regelmäßiger Zeitintervalle (etwa „durchsatz pro Sekunde“) oder über den gesamten Zeitraum akkumulierte Daten (etwa „Gesamtdurchsatz“), mit denen man das Verhalten eines Netzwerks in Abhängigkeit eines bestimmten Inputparameters (Ausgaben mehrerer Simulationsläufe mit einem jeweils leicht verändertem Parameter) darstellen kann.

Auf http://www.welzl.at/tools/ns/ sind zwei entsprechende einfache Programme verfügbar: throughput und stats.

Throughput

Möchte man sich z.B. die empfangenen Pakete des Typs „cbr“ pro Zeitintervall (standardmäßig 1 Sekunde; Anmerkung: das Intervall kann direkt im Quelltext angepasst werden) im Tracefile „out.tr“ ausgeben lassen, so erledigt das Throughput mit dem folgendem Aufruf:

```
perl throughput.pl out.tr cbr
```

Das Ergebnis ist das Folgende:

```
1       219000
2       241000
3       248000
4       249000
5       133000
```

Die Simulation dauert also 5 Zeitintervalle, die in der ersten Spalte aufgelistet sind. In der zweiten Spalte ist die Summe der empfangenen Pakete innerhalb des Intervalls dargestellt. Die Ausgabe kann zur weiteren Verwendung noch in eine Datei z.B. out.dat umgelenkt werden. Dies erledigt man mit dem folgenden Befehl:

```
perl throughput.pl out.tr cbr > out.dat
```

Das Ergebnis kann dann mit gnuplot visualisiert werden (näheres siehe Anhang B und in der Dokumentation zu Throughput). Eine Einschränkung auf bestimmte Ziel- bzw. Quellknoten ist durch die Angabe zusätzlicher Parameter möglich. So betrachtet:

```
perl throughput.pl out.tr cbr 2
1       111000
2       125000
3       125000
4       125000
5       64000
```

ausschließlich Datenströme, die den Knoten 2 als Ziel haben. Der Aufruf:

```
C:\trace_bak>throughput.pl out.tr cbr 2 1
1       111000
2       125000
```

```
3       125000
4       125000
5       64000
```

schränkt die Auswertung zusätzlich nur auf Verbindungen mit Knoten Nummer 2 als Ziel und Knoten Nummer 1 als Quelle ein.

Stats

Das Programm Stats gibt eine kurze Statistik über sämtliche empfangenen bzw. verlorenen Pakete sowie das Queue-Verhalten der Simulation aus. Dabei werden in folgendem Beispiel sämtliche Zeilen mit Pakettyp cbr in der Datei out.tr herangezogen, deren Ereignisse „r“ für empfangene Pakete bzw. „d“ für verlorengegangene Pakete sind.

```
perl stats.pl out.tr cbr
Total Received:         1090000
Total Lost:             10000
Average Queue Length:   3.75
```

Analog zu Throughput lässt sich auch mit Stats eine Einschränkung auf bestimmte Ziel- bzw. Quellknoten vornehmen. Weitere Details sind der den Programmen beiliegenden Dokumentation zu entnehmen.